LA-UR-98-4531

Title:

PC DAQ
A Personal Computer Based Data Acquisition System

Author(s):

Gary Hogan

Submitted to:

PC DAQ

A Personal Computer Based Data Acquisition System

Gary Hogan

October 29, 1998

General Introduction

PC DAQ is a general-purpose data acquisition and replay analysis program. It can also be used as a control shell for Monte Carlo programs. It is being developed at the Los Alamos National Laboratory as part of the Proton Radiography project.

Program Objectives describes the original purpose for developing this program.

Program Description gives a general description of the program.

Running on other computer hardware is also possible, but not tested.

Objectives

We have several objectives in mind for the program.

1. **Replacement for the Q/VAX system.** The Q data acquisition system developed at LAMPF/LANCE is no longer supported in either its software or hardware components. We need a new DAQ system that is supportable.
2. **Software Licensing.** Many DAQ/Analysis systems use the CERN library as a source of programs for analysis and plotting. However, because of the military nature of the Proton Radiography project, use of the CERN library is not appropriate. We also want to avoid expensive, single purpose licenses such as LABVIEW® as it is anticipated that this program will be used on many platforms. PC DAQ provides its own graphics display. Plot analysis can be done either in code or in EXCEL.
3. **Computer Hardware.** We want to adopt a computer hardware platform that is powerful and inexpensive.
4. **DAQ Hardware.** Moderate amounts of CAMAC hardware have been inherited from the shutdown of LAMPF. A CAMAC based DAQ system could reuse this capital investment. We also want to read data in from a variety of sources.
5. **Programmer Expertise.** Because of the considerable time pressure to develop a new system, adopting a familiar programming environment would speed up the development process. A system modeled on the LAMPF Q system would make it easier for users at LANL to learn a new system.

PC DAQ meets these objectives. As a DAQ system, its general design is based on the Q system. It handles multiple types of triggers and uses the same syntax to define histograms and histogram tests. The code is developed in-house. The only licenses needed are for the languages Visual Basic 5.0 and DEC Visual Fortran 5.0. While some reinvention is involved in this procedure, it allows for tighter integration of the final product and provides for local support. The newer Pentium and Pentium Pro chips meet the hardware computer needs. Finally, the principle author (Gary Hogan) already has experience writing Windows/Visual Basic DAQ systems.

Description

PC DAQ is designed to run on a 32-bit Windows® operating system (Windows 95®, Windows NT 4.0®, or above) running on the INTEL® 80x86 processor line. It is written in a combination of Microsoft Visual Basic 5.0 (VB EE) and DEC Visual Fortran 5.0. The user interface is written in VB. Most of the analysis routines are in FORTRAN. For a regular DAQ program reading from CAMAC, only the FORTRAN analysis routines need to be altered by the user. If data is also to be acquired from Active X Servers, then the user will need to use the VB analysis routines also. Windows NT is the preferred system as some features are limited under Windows 95. NT also provides better security and handling of Active X servers.

The current DAQ codes uses a National Instruments AT-GPIB/TNT card and driver (part 776836-01) to talk to a Kinetics System 3988/G3A CAMAC crate controller. For 16-bit words, transfer speeds of up to 400 Kbytes/sec in DMA mode have been observed with this hardware combination. A knowledgeable

programmer should be able to convert the hardware portion of the code to other hardware within two to three weeks. Up to 7 CAMAC crates are supported in the current version, though the GPIB interface can handle more. Other GPIB devices such as scopes and pulsers can also be read into the data stream. Any Active X server can be used as a source of both data and triggers.

Event triggers can come from a number of sources. The main hardware trigger source is assumed to come from a LAMPF Event Generator CAMAC module. It is a small modification to use any CAMAC module that generates a LAM as the trigger module. Twenty-four (24) events are available (1 through 24). Eight hardware triggers (4 through 12) are provided via NIM input. Events can also be generated in the software from internal timers, manual buttons, Active X servers, and other by other events.

The program provides both DAQ and replay (disk file input) modes. Extensive software control flags are provided so that the user can control the flow of data through the program. Flags can be set either in script command files or interactively. Histogramming, testing, and plotting packages are provided. Histogram data can be exported to spreadsheets or analyzed in user supplied programs. Plots can be copied and pasted as bitmap objects into other Windows programs or printed. A run keyed database is provided. The program can also be remotely controlled over the Internet.

More details can be found in the comparison to Q.

Running on other computer hardware.

In principle, PC DAQ could run on any system hardware running Windows NT® to which Microsoft® has ported Visual Basic, and that have a Windows version of FORTRAN (such as DEC Alpha's®); but this has not been tested. A suitable hardware interface card and driver would also be needed.

Copyright

Quick Comparison to Q

PC DAQ is loosely based on the Q DAQ system developed at LAMPF. Where the Q system is a set of independent programs tied together by a global data section, PC DAQ is a set of programs tied together using the Windows DCOM technology. Both PC DAQ and Q commands can be entered from text command files (called scripts in PC DAQ). In Q, the user at a command prompt types interactive commands. PC DAQ interactive commands are entered though a Windows style Graphical User Interface (GUI). Commands are initiated by clicking the mouse on command buttons, check boxes, drop down lists, or by typing in text boxes.

Both programs support multiple hardware and software triggered events. Initiation of an event (a trigger) in hardware starts a user defined set of command codes that acquire data from CAMAC hardware or a Active X Server. In Q, the DAQ command codes are written in QAL. In PC DAQ, CAMAC command codes are written in FORTRAN using FCNA or GPIB subroutines. Active X data is read in using VB analysis programs. In both systems, the user may supply separate analysis routines for each event type to look at and use the acquired data. In Q, the routines are called PROCnn; in PC DAQ, they are called ANALnn or vbANALnn, where nn is the event number. The ANALnn analysis routines can generate and return various error and flow control information. Event enabling can be done either by command scripts or by interactive controls. Events can be in either may or must process states. In PC DAQ, ANALnn routines can be called repeated times for a single trigger to enable analysis of a buffer containing multiple events. Event triggers can come from hardware (CAMAC), internal timers, Active X servers, or other events. Data can be read in synchronously or asynchronously.

In Q, data logging (“taping”) is done by explicit calls to output routines. In PC DAQ, the actual logging is automatic, with the user supplying various command flags that can control logging at the run, event, trigger, and variable levels. In PC DAQ, output data can be a mix of integer and real data formats. User I/O to the logging media maybe added in the near future.

Q defines histograms by a text command language in HSU. PC DAQ uses an extended version of these commands with the same syntax. PC DAQ also has a GUI definition control for histograms. Like the taping case above, Q uses explicit calls to increment histograms, whereas PC DAQ does incrementing automatically with command flags to control what is incremented. Q allows only integer values to be histogrammed; PC DAQ allows both integer and real values for the X-axis. PC DAQ allows non-integer increment values. In PC DAQ, histogram sizes are limited only by the virtual paging ability of the NT operating system (the smaller of 2 gigabytes or the available space on your hard drive), though sizes above a few Mbytes would require adjusting array sizes by an expert in the program. Histograms can use either 4-byte integer or 8-byte double precision buffers. Future versions may remove the requirement for expert intervention for big histograms and allow the user to call the histogram increment routines. Both systems provide 1 and 2 dimensional histograms. PC DAQ also supplies “time” histograms, which are histograms of a data value as a function of time. Time histograms allow the user to review data values over the last several events (up to 5000). PC DAQ does not support live dot plots (though it could if there was demand). Both systems allow for the selective or wholesale saving of histograms to disk. Histograms can be saved automatically at the end of each run as part of the data stream.

A test package very similar to the Q test package is provided. The test package defines tests that are used to see which histograms should be incremented. Tests can be defined in scripts or interactively (indirect gates and boxes). Test result summaries can be printed or copy/pasted into other programs.

Both systems provide plotting packages. In Q, the system is called HPL. In PC DAQ, the plotting package is based on the xyPlot package defined by Tom Mottershead. The plotting capabilities of the PC DAQ are rapidly improving. Currently, the user can control colors, line widths, and the visibility of most parts of a plot. Font control will come with a future version. Plots can be copied as bitmaps to other programs or printed. Multiple plots can be displayed in a single window. The user has control on how the plots are placed and/or overlaid. Plot scales can be locked together so that zooming on the control plot automatically rescales other plots. Plot analysis is currently limited to what the user may supply in analysis FORTRAN code. For analysis beyond that, histogram data can be exported to spreadsheets.

Both systems have a database utility. In Q, this is the PRM program that accesses a global data section. In PC DAQ, database values are read from an ASCII file(s) at the beginning of the run. The database is keyed to the run number, i.e.; the value of a database variable can depend on the run number. It is easy to define the PRM variables used in Q for legacy Q programs. A variable management system has been defined to make it easy to add new variables to the database and analysis code.

Software Organization

General

The program can be divided into two parts. The first part is the VB (Visual Basic) code that makes up the master control program. It can be run either within the VB development environment or as a stand-alone program. With the introduction of Version 5, VB code can now be compiled into native code, which increases the speed over the old interpretive modes in which Basic used to run It can still run slowly if all the checks are turned on. To prevent a bad speed penalty, disk I/O and computational intensive parts of the code are written in FORTRAN. The FORTRAN code is the second part of the program. Using FORTRAN for the analysis code also allows the migration of old code to the new platform. For the best speed, the disk I/O is done using System I/O calls rather than FORTRAN I/O calls. The FORTRAN is broken up into a number of Dynamic Link Libraries (DLL). A DLL is a library of routines that is loaded and linked to the main program only at run time. This means that a DLL can be compiled and linked separately without having to relink the entire program. The DLLs in PC DAQ are:

1. **FCNA**: This contains the code that talks to the GPIB drivers. It defines the FCNA commands. If a different interface were used, most of the communication changes would be done here.
2. **FileDLL**: This contains the code that does the disk I/O and histogram management.
3. **Q Test Package**: This contains Q test package code. Note that this code is a complete rewrite of the original package. Only the subroutine names are the same. From the user's viewpoint, the only change is that integer arguments to the subroutines are 4-byte integers instead of 2-byte integers.
4. **xyPlot**: This is a Windows® implementation of Mottershead's xyPlot subroutines. While generally following the original xyPlot specification, it has numerous additions and changes. Plotting is done using Windows' hardware independent graphics primitives, so plots can be done to the screen, printer, or other device with very little device dependent code. Currently, xyPlot is limited to only the single, built-in font This will be upgraded in the future. Other features maybe added later.
5. **User Routines**: This contains the user supplied analysis routines (ANALnn). This is the only DLL that the user should need to alter. Shell versions of the user routines are supplied.
6. **Database**: This contains the database I/O codes. The database is an ASCII orientated database keyed by run number.
7. **Utilities**: This contains various string functions converted from Basic.
8. **Scope Routines**: If you are talking to the Scope Monitor program, you will need this DLL
9. GPIB. We use Version 1.3 of the National Instruments GPIB DLL. The program does not work with Version 1. 4 for unknown reasons. A special, fake, version of the GPIB routines is provided in order to generate the .lib file the compiler needs to link to the DLL.

The working version of a DLL must be in the …\system32 subdirectory of the Windows directory. If you create a new DLL, you must copy the new version of the DLL to the system directory. If you have multiple versions of a DLL on the same machine, you need to exercise care that the version of the DLL in the system directory matches your task.

With this release of PC DAQ, the VB code is also divided up. This was done to enable remote control of the program over the Internet using Windows' DCOM technology. The pieces are:

1. PCDAQ.EXE: This is the GUI interface that the user sees and interacts. It has essentially all of the forms in the program. All users, whether remote or local, use this same set of forms. Thus, you have the same control over the program where every you are.

2. DAQControl.exe: This Active X server has the main guts of the program. The user analysis routines are called from this program. It also currently contains the VB user routines.

3. PCSystemInfo.exe: This Active X server is a utility program used to get information about the computer system and user.

It is a feature of VB that it can call FORTRAN programs that are in DLLs, but FORTRAN programs cannot call VB. Also, while one DLL can call routines in another DLL, two DLLs cannot both call each other, i.e., only one DLL of a pair of DLLs can be dependent on the other. This leads to a hierarchical structure for the program.

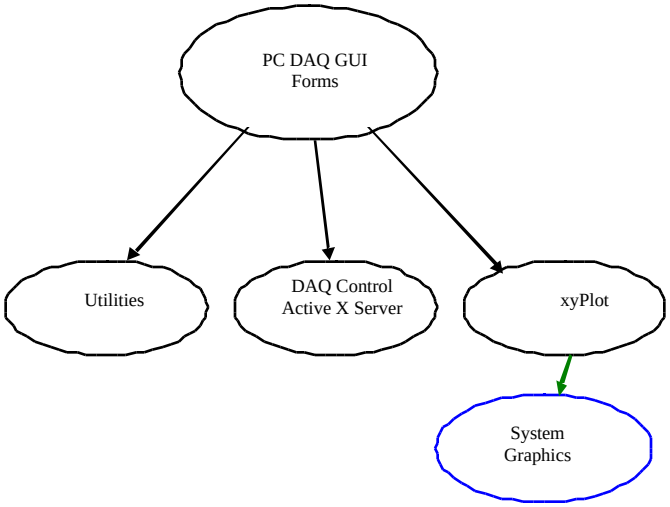


Figure 1, GUI Layer

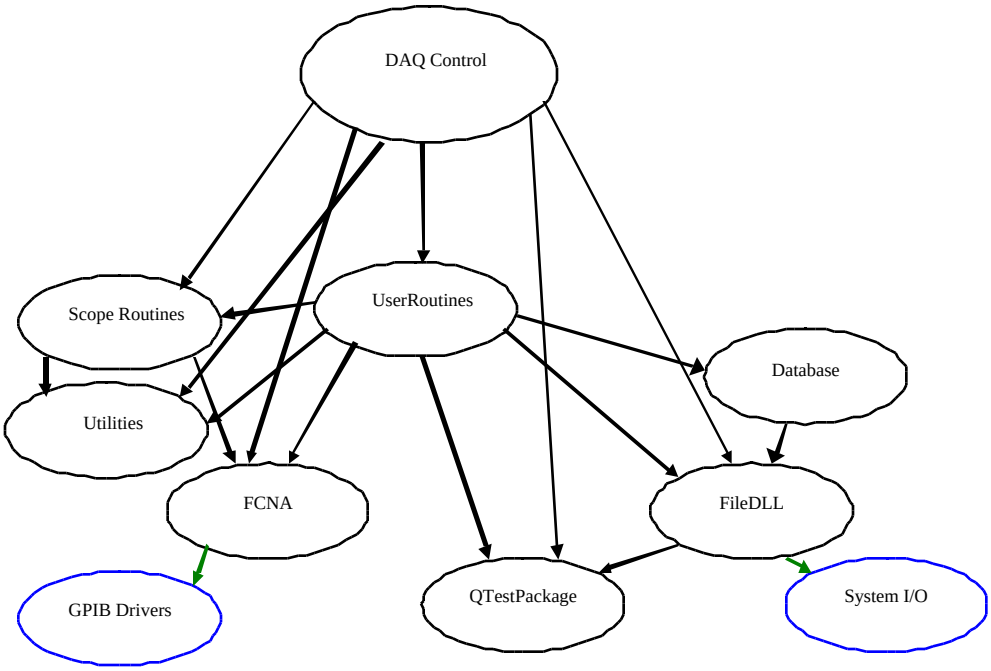


Figure 2, DAQ Active X Server

One of the requirements for calling DLL routines is that the calling program has to be given a description of the subprogram. This includes the subprogram name used by the user in writing code, the name used by the linker, and a detailed description of the argument list. This is done with DECLARE statements in VB and INTERFACE statements within FORTRAN. You will find files (usually with 'IMPORT' included in the file name) for the DLL being used. In general, I have put only the subprogram definitions that I am using within the INTERFACE/DECLARE files, so they are not complete. Making them complete is a low priority project. My FortranToBasic utility program can be used to convert a FORTRAN Interface block into VB Declare statements.

DEC's FORTRAN is a FORTRAN 90 compliant compiler with support for FORTRAN 77. The FORTRAN code is written in F77 format (72 columns, column 6 as continuation, etc.). Because it is sometimes the only way to do thing, or just much easier, some of the code uses FORTRAN 90 syntax. To make communication with VB possible, type structure definitions must always have the SEQUENCE command. The compiler option for Structure Element Alignment must be set to 2. Almost all the code uses IMPLICIT NONE (FORTRAN) or OPTION EXPLICIT (VB) to force type declarations for all variables.

Where is the Data?

Another feature of a mixed language environment combining VB and FORTRAN is the lack of common data storage. There is currently no way for VB to get to a FORTRAN common block directly. Thus a decision must be made about where data is to be stored, VB or FORTRAN. Different decisions were made for different types of variables. In some cases, data is duplicated.

Data Blocks

Data blocks are arrays of numbers used for I/O buffers, data results, and histogram input. Only three blocks are currently supported. They are the "System Blocks." The first block is a large 16-bit integer array [IntegerData()]. The second is a 32-bit single precision real array [SingleData()]. The third is a 32-bit integer array [LongData()]. All I/O is done from these blocks. Currently, histogramming can only be done from these arrays. The sizes of these arrays are defined by the parameters NumberOfIntegers, NumberOfSingles, and NumberOfLongs. As with many of the parameters in the program, these exist in

both the VB and FORTRAN code. All three arrays must have the same size. The data arrays themselves exist only in the FORTRAN code. VB communicates to these arrays through subroutine calls.

The array common blocks are defined in PCDAQBuffers.inc. The include file EGEOM.FI divides up the system blocks into a variety of arrays and scalars if you are using to the variable management tool. That database code allows you to initialize these blocks.

There is a NULL data value defined for each system block (a large negative number). At the beginning of every run (or batch of runs if in auto restart or replay batch mode), the system blocks are cleared to the NULL value before any user routines are called. NULL data is not displayed or histogrammed. It is written to "tape" if present. Because the NULL data values have a large absolute value, using variables without first defining their values can easily lead to "Overflow" errors. If you start getting such errors, start looking for uninitialized variable usage.

The current contents of the data blocks can be viewed using the Data Screen. Click on Data Screen under the View Menu. The labels that you see with the values are defined in a LABEL script. For those labels with associated time histograms, the past 5000 values can be viewed if the Data Screen is in histogram mode. Because this screen is a convenient way of looking at the data, users are encouraged to equivalence their working variables to one of the three data arrays and define a LABEL for each variable.

The user with the LABEL script describes the structure of the System Block. This script tells the program how the locations in the blocks are to be associated with events and how they should be displayed. It also flags which variables are to be logged to disk. The CAMAC I/O buffer definitions are given by the LABEL script commands.

Histogram Data

Histogram data comes in two parts. The first is the histogram definition. In the VB code, definitions are stored as histogram class objects within a collection. In FORTRAN, definitions are contained in type structures HistInfoLabels and HistInfo. These structures are used to define the arrays HistogramNames and HistogramList. The first contains the name strings (title, etc.), the second all other histogram information. This information is duplicated in VB and FORTRAN codes. The only difference is that the names are truncated to 40 characters in FORTRAN. In FORTRAN, the arrays are in PCDAQBuffers.inc.

The second part of the histogram data is the actual frequency data. For 1D and 2D plots, the data is stored in either the 32-bit integer array HistBuffer or the 64-bit double precision array HistDoubleBuffer. For time histograms, the information is in TimeArray, and TimeHist. These arrays only exist in the FORTRAN code. VB communicates to these arrays through subroutine calls.

All of the above is subject to change. I am considering a major change in the near future where the histograms (definition and data) become C++ objects. Then data array space would only be allocated on an as needed basis. The number and size of histograms would be limited only by the (virtual) memory space of the machine. Communication to the histogram objects would be by subroutine call for both FORTRAN and VB.

Test Package

Test package data is all stored in the FORTRAN code. VB gets at that data via subroutine calls. Note that internally, the test package data structures are completely different from the Q version. VB organizes the test definitions as a collection of class objects.

Control Data

Most control data resides exclusively in the VB code. Most global variables are defined in PCDAQGlobal.bas. Only those flags that must be available to the FORTRAN code have been duplicated. Most numeric control variables have a list of defined values encoded into parameters. For example, the null data values have the parameters NullDataI2, NullDataR4, etc., defined. These parameters are often duplicated in both the VB and FORTRAN code.

Also among the duplicated information is the numeric portion of the LABEL definitions. The string part of the Label definition is not duplicated. As with the histogram definitions above, the label definitions are coded as class objects. If you are using the variable management tool, some of the label information (including the name and comment) can be found in the EGEOM database structure (EXPER_GEOM.FOR).

Where is the Code?

Setting up Subdirectories

The code is located in seven subdirectories. There is one sub-directory for each FORTRAN DLL and one for the VB code. Independent utilities such as RemoveFiles are in other directories. It is assumed that you have set up your code so that all the subdirectories are under the same master directory. The FORTRAN subdirectories are ..\FCNA, ..\FileDLL, ..\UserRoutines, ..\xyPlot, ..\Database, and ..\QTestPackage. Microsoft calls each of these DLLs a "project." The project definition file has the extension ".dsw". Each directory contains the source code and project definition files for that DLL. Changing these names is going to cause you more trouble than its is worth, so don't do it. As Figure 1 shows, these DLLs call other DLLs. Those DLLs that call other DLLs have a reference to the appropriate .LIB file in their Linker control line. These .LIB entries must point to the actual location of the .LIB file for that DLL. They do this by using the double dot (..) notation in the path portion of the file name. Thus the release version of the FileDLL linker command lists "..\QTestPackage\Release\QTestPackage.lib" as an input file. The debug version links to "..\QTestPackage\Debug\QTestPackage.lib".

Below each DLL source directory are subdirectories called ..\Release and ..\Debug. These subdirectories are created when you compile and link the DLL. Each sub-directory contains the object modules and linked DLL for the configuration selected. Two configurations are possible, Debug and Release. The debug version contains information and checks used by the debugger for annotating code. The release version does not have this and runs faster. When you change a DLL, you need to copy the updated .DLL file to the ..\system32 directory.

The VB code is currently located in ..\VBCode. Along with the VB code and executable, this is the default directory for script files. Database files are usually under the ..\DatabaseFiles sub-directory. I find it convenient to also defined a ..\OutputFiles directory for data and histogram files. Help source files are in the ..\PCDAQHelp sub-directory.

CAMAC (FCNA)

A number of routines have been defined for use by the user to talk to CAMAC. Along with carrying out the assigned CAMAC function, the FCNA commands below also keep track of the state of the GPIB crate controller state settings. If instead of using the FCNA subroutines, you use the GPIB commands IBWRTF and IBRDF, be sure you also update the crate controller state variables. Such code should only be written by experts. The routines in FCNA are self-contained and can be used by other programs that want to talk to CAMAC.

These routines are in the FCNA DLL. Arguments are 4-byte integers (Integer*4) unless noted otherwise.

Call **FCNA** (F, C, N, A, Lun, Wrt24, Value, Q, X, Err( )) :
F : CAMAC function code
C : CAMAC crate
N : Slot
A : Address
LUN : Undefined. Kept for compatibility.
Wrt24 : = -1 (true), use 24 bit write, otherwise 16 bit write.
Reads are always 24 bits.
Value : Value to read/write to CAMAC
Q : Q returned by module
X : X returned by module
Err(2) : Err(1) = 1 => Success, otherwise failure.

Note that the sign of this error flag is opposite
of what is used in the rest of the programs.
Err(2) : Undefined.

ReturnCode = **ControlFCNA** (F,C,N,A,Q,X) :

Issue CAMAC control function (F = 8 → 15, 24 → 31)
F : CAMAC function code
C : CAMAC crate
N : Slot
A : Address
Value : Value to read/write to CAMAC
Q : Q returned by module
X : X returned by module

A return code of -1 is success. The parameter "Success" is defined in CamacCommon.inc.

ReturnCode = **ReadFCNA** (F,C,N,A, Nbits, Value, Q,X) :
Do a CAMAC read.
F : CAMAC function code
C : CAMAC crate
N : Slot
A : Address
Nbits = 8, 16, or 24.
Value : Value to read from CAMAC
Q : Q returned by module
X : X returned by module

A return code of -1 is success. The parameter "Success" is defined in CamacCommon.inc.

ReturnCode = **WriteFCNA** (F,C,N,A, Nbits, Value, Q,X) :
Do a CAMAC write.
F : CAMAC function code
C : CAMAC crate
N : Slot
A : Address
Nbits = 8, 16, or 24.
Value : Value to write to CAMAC
Q : Q returned by module
X : X returned by module

A return code of -1 is success. The parameter "Success" is defined in CamacCommon.inc.

INTEGER*4 function **IBWRTF** (ud, buf, cnt)
INTEGER*4 ud ! GPIB device identifier
Integer*1 buf(*) ! Output buffer
Integer*4 cnt ! Byte count

INTEGER*4 function **IBRDF** (ud, buf, cnt)
INTEGER*4 ud ! GPIB device identifier
Integer*1 buf(*) ! Output buffer
Integer*4 cnt ! Byte count

At some future time, I will provide subroutines for more complicated CAMAC operations such as a DMA read of a LeCroy® FERA Memory Module.

FileDLL

Most of the routines in FileDLL are used for (a) disk I/O, (b) VB-FORTRAN information passing, and (c) histogram management. The user would normally only be interested in the last item. These routines will

be discussed later in the section on histograms and examples. The first two topics will be left for a future report.

xyPlot

At this time, there is no reason for a user to be calling these routines. Hypothetically, if a provision was added for drawing an event display, then access to these routines would be useful. There are no plans at this time to provide hooks for such a feature (though it would not be a bad idea). For the moment, anyone interested in using these routines should look at Mottershead's documentation on xyPlot. They can be used from any Windows programming environment (VB, C++, FORTRAN, VBA [i.e., Excel, Access]). Updating the existing documentation of xyPlot to include my changes and extensions is a low priority project awaiting an expression of interest.

Q Test Package

This is an implementation of the Q Test Package as defined in MP-1-3412-3, March 1986. Because I recoded from scratch, I have made some changes. The internal structure of the database is completely different, but the user should not see this. The major change for the user is that all arguments in the calls are Integer*4, not Integer*2. Also, the call to TSTEXE (the test call) as been upgraded so that the data array can be any type (I*2, I*4, R*4, or R*8). Indirect gates and boxes can be defined using the mouse on histograms. The test database can be dumped in binary form (for replay) or text suitable to be read into Excel. Test setups can be saved to scripts to save the indirect gate/box settings for later use.

Note that unlike data logging and histogramming, tests are NOT done automatically. The user must include calls to TSTEXE in their code to make the tests happen. Test definition formats can be found in the section on Scripts. Otherwise, I note that the two important calls are:

CALL **TSCLFB** (BlockNumber, IERR) ! Clear test block flags

CALL **TSTEXE** (BlockNumber, DataArray, IERR) ! Do block tests

Note that the block number here has nothing to do with the system data block number used for defining labels and histograms. It refers only to the test block number defined in the test script.

Control

Program flow control is all done in the VB code. FORTRAN routines are only called to carry out specific functions. The FORTRAN routines effect the flow of the data only to the extent that they can return certain exception control values. The VB master program then reacts to these exceptions. The exceptions that the user is most interested in are those generated by the ANALnn routines. These exceptions can, among other things, cause (a) the routine to be called again for the same trigger, (b) abort data logging or histogramming, or (c) abort the entire run. This is discussed in detail in the analysis section.

A complete discussion about how the VB code is organized is beyond the scope of this report. I discuss here only the subject of class usage. The VB code as many class structures. Some define objects, other define collections of objects. An example is the PLOT class. This object contains the information and code to display a single histogram. This is where line type and font information is stored. The GRAPH class is part of the NOTEPAD MDI child form. A GRAPH is a collection of PLOTs. The GRAPH class information contains the page title and plot list. Other classes that are in the VB code are part of canned utilities routines. Labels, tests, and histograms are classes.

User Routines

Most user changes will be to the ANALnn routines. These are part of UserRoutines.dll. Each routine is a function. The argument of the function tells the routine which of several different operations related to an event type should be performed. The most common operations are to read CAMAC and to analyze data. Thus, the equivalent of both Q's QAL code and PROCnn code are in the same routine in PC DAQ. Details of the operations are given in the section on analysis.

The return value of the ANALnn function tells the control program what it should do next. The commands range from “proceed normally” to “stop all data acquisition.” The return value is a bit-mask, so various commands can be combined. Again, details of the defined return commands are given in the section on analysis.

Shell versions of the ANALnn programs are supplied. The shells contain INCLUDE statements for most of the system common blocks to which the user needs access. They also provide a code structure to handle the various operations the program is requested to handle. If you are not familiar with the SELECT CASE programming structure, I suggest looking it up in the extensive on-line help provided with FORTRAN as an exercise for the student.

One of the INCLUDE file is AnalysisCommon.INC. The first part of this include file contains vital information such as the location for the event trigger module and run number. Users are on their honor not to mess with this information or the control parameters in other INCLUDE files. Other than that, the user may use this INCLUDE file to defined such things as the slot and crate locations of CAMAC modules, variable equivalencies to the system blocks, and other information that should be shared by several ANALnn programs. The variable management tool is useful here. The user is free to add other routines and include files to this DLL. Note that Microsoft prefers .fi as the extension for FORTRAN include files rather than .inc. Either will work, but .fi files are given a nicer editing environment.

How do I make changes?

The examples below assume you are changing code in UserRoutines.dll. Similar comments apply if you change other DLLs. And remember, when in doubt, read the manual. The complete FORTRAN manual plus much, much more is available in the on-line help files. FORTRAN PowerStation automatically creates, maintains, and uses MAKE files for fast compilation of code.

Opening the Project

FORTRAN code is edited and compiled by the DEC Visual Fortran® program. Start this program. You pick a project to open from the File menu. Previously opened projects are listed at the bottom of this menu. If UserRoutines is not listed, use the OPEN WORKSPACE menu command to find and open PCDAQ.dsw.

FORTRAN Help Information

On the left side of the screen is a area which usually displays one of three lists. You select which list by clicking the tab at the bottom of this area. Class View is usually empty. File View lists all the files in the project. InfoView lists the current set of on-line books. Other book sets can be viewed by choosing a different set from the drop down list in the tool bar. If you get a chance, browse these books. Its all there, including examples. You can also access these help books for the standard Help menu command.

Opening and Editing a File

From the file view on left side of the screen, double click the file you want and it will be opened for you. If all you see is “UserRoutines Files,” double click on this to expand the list. Include files can be found by expanding the Dependencies heading at the bottom of the list. Standard Windows text editing commands are used to edit files.

Compiling and Building.

Directly under the Build menu command are the Compile xxx.for and Build UserRoutines.dll commands. The first command saves, then compiles, the routine in the currently active text window. The Build command saves and compiles are files that have changed since the last build. It then builds a new DLL file. To use this file, you need to move the created DLL file to the ..\system32 directory.

Test runs from within FORTRAN.

To run the program without leaving the editing environment, select the Debug or Execute command under the Build menu. If you have not already defined the main executable, it will ask for the file. Give the complete path and filename for the location of PCDAQ.EXE, including the extension. If you mess up the name, use the menu path BUILD / Settings… / Debug to get to the text box "Executable for debug sessions" and put in the correct information. If you are running in Debug mode, it will complain that there is no debug information for PCDAQ.EXE. That is okay. The debugger will still work on the DLL code you are testing. I find the debugger to be quite nice and I encourage you to use it. Remember, each time you change the program, the new DLL needs to be copied to the ..\system32 directory.

Known problems

Information Duplication

Several types of information, such as label and histogram definitions, are stored in both the VB code and FORTRAN code. This can lead to synchronization problems, particularly if the user fools with forbidden items such as the histogram definitions. In general, control and definition changes should only be made from the user interface, not in analysis code. Some of these temptations will be removed in future versions. All I can ask is that you be careful.

FCNA and UserRoutines

The optimizer in FORTRAN has some problems working with the GPIB DLL. FCNA and UserRoutines will bomb in DAQ mode if the code is optimized. UserRoutines does seem to work in replay mode if optimized. So some care needs to used in selecting the compilation conditions of these DLLs depending on the mode you are running under. The code does not work with Version 1.4 of the National Instruments GPIB DLL. Use 1.3 or earlier.

Speed

Because the program flow control is in VB, it can sometimes get bogged down in a high rate environment. For the Proton Radiography experiment, we use a feature in the analysis that allows an analysis routine to be repeatedly called to step through the sub-triggers stored in a FERA Memory buffer. Normally, control would be returned to the VB program after each sub-triggers is processed. This allows the control program to interrupt processing in favor of new data (may process). In replay mode, this feature is not needed and returning to the VB program after each sub-trigger slows down analysis. Therefore, techniques have been developed for calling directly between ANALnn routines. Talk to me if you need these methods. In the Proton Radiography experiments, the database command EGEOM SPEEDFLAG has been defined to toggle between these modes.

DAQ Properties

CAMAC

Functionality

The preferred method for the user to talk to CAMAC is to use the FCNA commands provided in the FCNA DLL. These routines handle both the GPIB command codes and the crate controller commands. This is usually all the user needs to know.

If you need to talk to the crate controller, a number of canned programs exist in FCNA to perform common functions. You would probably use these only if you were writing a stand-alone program, say a module test program. Normal order of use would be:

1. Setup CAMAC with SetupCamac function.
2. Do a bunch of FCNAs with FCNA subroutine.
3. Close out with CloseCamac subroutine.

If you are writing a stand-alone program (assuming it is in FORTRAN), you need to:

Include the FCNA.LIB file in your project linker input list.

Include CamacCommon.inc and FCNAImport.inc in your main program. Either include the path to CamacCommon.inc in the INCLUDE statement or add the path under / Build / Settings… / FORTRAN / Preprocessor / INCLUDE and Use Paths.

After the above includes, add the line:
```
!MS$attributes DLLIMPORT, Alias:'_CAMACCOMMON' :: CamacCommon
```

Make sure the compiler option for Structure Element Alignment is set to 2.

Make sure the compiler option for optimization is OFF

SUBROUTINES:

ByteBreak (InputNumber, Buffer( )[I*1]) : Breaks out the lower three bytes (by value) of InputNumber and stores them in the byte array Buffer. Order is high, mid, low.

CloseCamac : Takes CAMAC controller off-line from GPIB bus and releases GPIB software from memory.

FCNA (F, C, N, A, Lun, Wrt24, Value, Q, X, Err( )) : (See also the ReadFCNA, WriteFCNA, and ControlFCNA functions below)

F : CAMAC function code
C : CAMAC crate number (User assigned number set by SetupCamac)
N : Slot
A : Address
LUN : Undefined. Kept for compatibility.
Wrt24 : = -1 (true), use 24 bit write, otherwise 16 bit write.
Reads are always 24 bits.
Value : Value to read/write to CAMAC
Q : Q returned by module
X : X returned by module
Err(2) : Err(1) = 1 => Success, otherwise failure.
Note that the sign of this error flag is opposite
of what is used in the rest of the programs.
Err(2) : Undefined.

Call **FCNAErrMess** (Num, Message) . Converts an error return code from a number to a text string.
Num = Return Code. Note, From FCNA itself, call with -NUM.
Message = Output string with text of error

Other Functions:

ByteMerge (Buffer()[I*1]) : Returns an I*4 number made up of first three bytes in Buffer (a byte array). Order is high, mid, low.

CAMAC FUNCTIONS:

The following functions are used to talk to the Kinetic System 3988/G3A GPIB CAMAC Crate Controller. They return a ReturnCode, values of which can be found in CamacCommon.INC. Success is a value of -1.

Common Arguments are:

CrateNumber: User assigned crate number for controller. Defined by call to SetupCamac. When you call SetupCamac, you supply an array of GPIB hardware addresses. Crate #1 is the first entry in the hardware address array, crate #2 is the second array entry, etc.

Value: An I*4 value that is written or read from controller or module register.

Nbits: Number of bits to transfer. Values are 8, 16, or 24.

Functions:

CheckSwitch (CrateNumber) : Success means manual switch on controller is On-line.

ClearInhibit (CrateNumber) : Clear inhibit issued by crate controller.

ClearUseQX (CrateNumber) : Turn off the return of the status byte (Q and X) [SBE].

ControlFCNA (F,C,N,A,Q,X) : Issue CAMAC control function (F=8->15,24->31)

ReadControlRegister (CrateNumber, Value) : Read 3988 control register

ReadFCNA (F,C,N,A, Nbits, Value, Q,X) : Do a CAMAC read.

ReadLAMRequest (CrateNumber, Value) : Read LAM request register in controller.

ReadTransferCount (CrateNumber, Value) : Read Transfer Count register in controller.

SendC (CrateNumber) : Have controller issue a C clear.

SendZ (CrateNumber) : Have controller issue a Z clear.

Set16Bits (CrateNumber) : Set Controller to 16 bit transfers.

Set24Bits (CrateNumber) : Set Controller to 24 bit transfers.

Set8Bits (CrateNumber) : Set Controller to 8 bit transfers.

SetAddressScan (CrateNumber) : Set controller to Address Scan mode.

SetInhibit (CrateNumber) : Set inhibit issued by crate controller.

SetQRepeatScan (CrateNumber) : Set controller to Q-Repeat Scan mode.

SetQStopScan (CrateNumber) : Set controller to Q-Stop Scan mode.

SetSingleTransfer (CrateNumber) : Set controller to Single Transfer mode.

SetupCamac (NumberOfCrates, HardwareAddressList(), IssueCZ) : Sets up GPIB interface and CAMAC controller.

NumberOfCrates : Number of entries in list of Hardware addresses. Maximum size is defined by parameter NumOfCrates in CamacCommon.inc.

HardwareAddressList() : List of GPIB hardware primary addresses. Controller addressed by entry 1 becomes Crate #1. Empty entries are allowed. Later attempts to address an empty entry will generate an error code.

IssueCZ. If non-zero, C and Z are issued as part of the setup.
Program does the following as part of its setup:

Bring in GPIB-32.DLL. Routines that talk to GPIB board.
Get software addresses for controllers.
Issue GPIB clear to controllers.
Set the controller(s) to default values:
Inhibit cleared.
24 bit transfers.
Q and X (status byte) not returned.
Single transfer mode.
Check that controller(s) have their manual switch on-line.
Disable all LAMs in crate(s).
Do a Z if IssueCZ non-zero
Do a C if IssueCZ non-zero
Clear Transfer count register.
Clear SRQ enables.
Set control variables in CamacCommon.inc.

If an error occurs during setup, the GPIB interface is shutdown and the DLL released.

SetUseQX (CrateNumber) : Turn on the return of the status byte (Q and X) [SBE].

WriteControlRegister (CrateNumber, Value) : Write to 3988 control register

WriteFCNA (F,C,N,A, Nbits, Value, Q,X) : Do a CAMAC write.

WriteLAMMask (CrateNumber, Value) : Write to LAM Disable Mask.

WriteSRQMask (CrateNumber, Value) : Write to SRQ (GPIB Service Request) Enable Mask.

WriteTransferCount (CrateNumber, Value) : Write to Transfer Count register in controller.

GPIB ROUTINES

These routines talk to the GPIB software. They are not for the causal user. See National Instrument's documentation for 488 functions for details of the ibxxx functions. If you use these functions to directly change the state of the crate controller, be sure you also change the value of the appropriate state variable in CamacCommon.inc. Many useful parameters for these routines are in GPIBConstants.inc.

I*4 Function **LoadDll** (dummy) : Loads GPIB-32.DLL. Success = -1 return value.

Subroutine **FreeDll** : Frees GPIB-32.DLL.

I*4 Function **IBDEVF** (...) : Calls ibdev function.

I*4 Function **IBCLRF** (...) : Calls ibclr function.

I*4 Function **IBONLF** (...) : Calls ibonl function.

I*4 Function **IBWRTF** (...) : Calls ibwrt function.

I*4 Function **IBRDF** (...) : Calls ibrd function.

I*4 Function **GETIBERR** (...) : Returns iberr value.

I*4 Function **GETIBSTA** (...) : Returns ibsta value.

I*4 Function **GETIBCNTL** (...) : Returns ibcntl value.

You can also talk to the crate controller using GPIB commands (ex., IBWRTF).

Speed

Both bench test and actual experimental floor tests have been done on the speed of the hardware. We use a PC-clone computer talking to a GPIB CAMAC controller. The computer is a 166MHz Pentium PC running Windows NT 3.51. The GPIB interface is a National Instruments GPIB/AT-TNT board. The CAMAC controller is a Kinetics System 3988/G3A GPIB controller with DMA capabilities. The bench test program is written in FORTRAN using the FCNA DLL mentioned above to communicate with the National Instruments device software. A LeCroy 4302 FERA memory module is used as the data source. Time is measured using the SECNDS library function, which is good to 0.01 seconds.

Data is readout using 16-bit (2-byte) CAMAC reads. Rates are measured for various numbers of bytes in the memory. For a given number of bytes, the memory is readout 100 times and the average is reported. The memory is readout in LIFO (Last In, First Out) mode. The data comes out of the controller as a byte stream. For microchip based computers, the byte order is swapped compared to the internal byte order for 16-bit integers. Most of the measurements were made with code in place to correct the byte order. At the highest rate, a measurement was also made with the byte swapping turned off. Results and straight line fits are shown below:

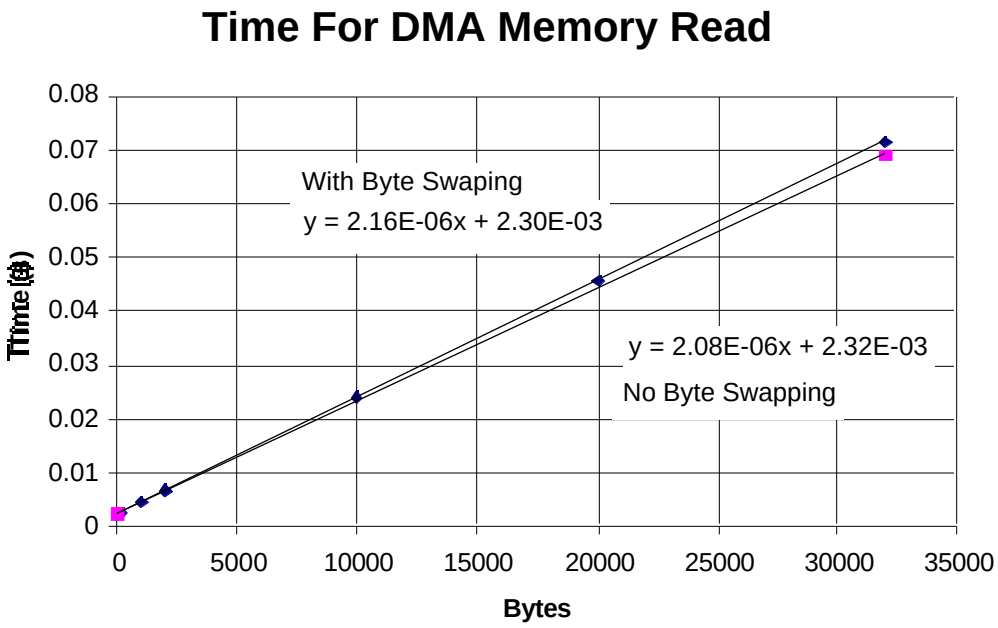


Figure 3, DMA Timing

There is an 2.3 millisecond overhead time in setting up the DMA transfer. DMA transfer rates are then 463 Kbytes/s to 480 Kbytes/s, depending on how the byte swapping is handled. The setup time is close to minimal. Tests were also done on a 66MHz 486DX2. There the results were 12 milliseconds setup time and 380 Kbytes/s to 470 Kbytes/s DMA speeds. So the DMA speed is almost independent of processor speed while the setup time scales with the processor speed. Note that CAMAC's maximum DMA speed is 2Mbytes/s in 16-bit mode.

The Pentium computers have been used in experiments at Brookhaven with the full PC DAQ software. When you add in time to write data to disk and overhead from the PC DAQ program, the effective speed drops to around 330 Kbytes/s. Taping can reduce this by another 10%. We may be able to get some of the loss due to taping back by having the backups done by another machine over a fast (100BaseT) network connection.

The setup time of 2.3 milliseconds comes from the number of GPIB command cycles it takes to set up the DMA transfer. Each GPIB command takes about 0.4 milliseconds to complete. For comparison, a single word read takes 2 GPIB command cycles, and so takes 0.8 milliseconds to make. For slower machines,

you should read the data out only when the memory is close to full, otherwise you will be dominated by the setup times. Reading out CAMAC modules in a non-DMA mode is a slow process. The KS controller does have a readout mode (Address Scan) that can read several non-DMA modules using a single GPIB command cycle. If enough modules are read, a speed increase of a factor of 10 over the non-DMA speed is possible.

Reading out data within the LAMPF 8 ms gap between macro pulses is not possible without overlapping the next pulse(s). So when running at LAMPF, you need to set up the data accumulation to take in several beam bursts for each readout cycle to reduce the dead time. You can also try flipping between two parallel memory modules.

I/O

CAMAC I/O should be done to the integer system data blocks. 16-bit data goes into IntegerData( ) and 24-bit data goes into LongData( ). The location for an event's I/O buffer is defined by a Label command with the name having the specific form: CAMACnn, where nn it the event number. Use a leading zero for single digit event numbers. See the section on label script formats for details on available options. The label command should define the maximum length buffer needed and whether the buffer should be automatically logged. Automatic logging can be overridden on a trigger by trigger basis using the ANALnn return codes. Data is placed in the I/O buffer by user supplied FORTRAN code in the appropriate ANALnn program. ANALnn does a CAMAC read when it is called with the operations argument set to "uGetTrigger." The location of the I/O buffer is available to the user via the array IntegerEventList (EventNumber) or LongEventList (EventNumber). This is the index of the first word of the buffer in the appropriate system block. For example, to set the third word of event 5 to the value 10, use:

```
IntegerData (IntegerEventList (Anal05Ev)+2)  = 10
```

Once all the data is read, set the counter IntegerEventSize (EventNumber) [or LongEventSize] to the number of words used. The program will use this counter to decide how much of the event buffer to write to disk. The user is responsible for insuring that the end of the buffer is not overrun.

I/O buffers are best-defined using the variable management tool.

In principle, you can use the real system block as the I/O buffer, but that is probably bad programming practice.

Control

This section has not been written yet.

There are separate control forms for:

1. Run Setup Mode
2. Data Viewing
3. Graph Definition
4. Plot Definition
5. Event Definition
6. Histogram Saves
7. Replay Data Viewing
8. Script Execution
9. Test Package Definition
10. Histogram Definition
11. Label Definition
12. Test Results Viewing
13. GPIB, CAMAC, and Database Interface
14. And on and on…

Plotting

A distinction needs to be made between histograms and plots. A histogram is a set of data, usually the frequency that some value has occurred. A plot is a graphical display of the data contained within a histogram. Several different plots can be defined for one histogram data set. A collection of plots on a single page or window is called a graph.

Defining a New Plot

To define a new plot, you must first open a window to display the plot. To open the window, use the menu combination File / New Plot. This will open a window which will have the text "[GRAPH]" in it. You now need to pick which histogram to display in this window. Use the menu combination Plot / Define... to bring up the Graph Definition form. This form has three large boxes on it and several controls. The big box across the top is where you may enter an optional page title. The list box on the left contains a list of all the currently defined histograms. The list on the right is the list of histograms that will be displayed on the graph page you are defining (the plot collection). From the left list, pick one histogram by clicking on the name with your mouse. The entry for each histogram contains 3 numbers and a name. The numbers are the event number associated with the histogram, the system block for the histogram (1=I*2, 2=R*4, 3=I*4), and the histogram type (time=0, 1D=1, 2D=2). Together with the name, these four pieces of information should uniquely identify a histogram. Once you pick a histogram, click the "Add →" button. The histogram will appear in the right hand list. If you picked the wrong histogram, you can remove it by selecting it in the right hand list and clicking "← Remove." Once you have selected the correct histogram, click on "OK" at the bottom of the form. The program will then draw the plot in the window using the current histogram data set. Please note that while the graph definition form is open, no data is being taken.

Saving a Plot

Plots and the underlying histograms can be saved in a number of way. Available options include:

Saving the layout of the graph page to a script file. Like everything else in PC DAQ, there is a script file that describes how a graph and the plots within it are organized. By saving this script, you can later bring back the same graph with all of your layout changes and enhancements. To do this, first make the window you want to save the active window by clicking anywhere on it. Then use the menu combination File / Save (or Control-S) to save the script file. The file will be saved under its current name (the caption along the top of the window). If you have not already saved the file, a Save File control form will appear. Be sure you save the file using the extension .SCR. Please to not try to edit [Graph] and [Plot] scripts.

Saving the current view as a bitmap file. You can copy and paste a graph picture into another Windows program using the standard Copy and Paste commands. To do this, first make the window you want to save the active window by clicking anywhere on it. Then use the menu combination Edit / Copy (or Control-C). This will copy the picture to the clipboard. You can then Paste this image into most Windows programs (Word, PowerPoint, etc.). To save the bitmap as a .BMP file, Paste the image into Paintbrush, then save as a .BMP file.

Saving the current view as a postscript file. Bring up the Print Manager. Find a postscript printer and make it the default printer. Go through its property control until you find a "Print to File" option and select it. Then "print" the file. The system will ask you for a file name. To convert this into encapsulated postscript, use GhostView. Remember to reset your default printer to what is normal. A large 2-D plot can be huge (> 50 Mbytes).

Saving the current view in other formats. I don't have specific recommendations. I would suggest trying to get a multi-format viewer off the web somewhere. Some of these will read in .BMP files and then resave them as GIF, TIFF, or other formats. At some point I do plan to be able to produce WMF (Windows Metafile) formatted files directly. This will allow you to edit the plots (but not the data) in programs like PowerPoint. One way to get a completely editable plot in other programs is to first move the histogram data to a spreadsheet (see last item below), create a chart, then paste the chart object wherever you want it.

Saving the text summary. With each plot, there is a short text summary available that lists such information as boundaries and averages. You can view the text by toggling the "Text" item in the Plot menu. Once in text mode, you can use the Edit menu commands to "Select All" then Copy to the clipboard. Then you can paste and save it to any windows program you wish. To save the text summary directly, use the menu combination File/Save As… and pick the extension .SUM. The text summary can also be found at the end of the graph script file as a comment section.

Saving the underlying histogram(s) as a binary HSV file. To save the histogram data of plot(s) in the window as a binary file, use the menu combination File/Save As… and pick the extension .HSV.

Saving the underlying histogram(s) as a text file readable by spreadsheet programs. To save the histogram data of plot(s) in the window as a tab delimited text file, use the menu combination File / Save As… and pick the extension .TXT. If the graph contains multiple plots, each plot will be saved to a separate file with "_n" tacked on to the name. "n" is the plot number in the graph.

Coping a Plot to another Windows Program

To do this, first make the window you want to save the active window by clicking anywhere on it. Then use the menu combination Edit / Copy (or Control-C). This will copy a bitmap picture to the clipboard. You can then Paste this image into most Windows programs (Word, PowerPoint, etc.).

Recalling a Plot

To recall a graph window that has previously been saved as a script file, open the File menu. If the graph was recently saved, the file may be in the bottom list shown on the menu. If not, use the Open… command to find and open the script file.

Multiple Plots on a Page

You can have as many plots on a page as you wish. You add or remove plots using the Graph Definition form (menu Plot / Define…). You change the contents of the plot collection list using the "Add →"and " ← Remove" buttons. All plots added with the "Add →" button start out as full-page plots with a plain, black and white layout. You can move and resize the plots using the plot editor form. Simply click on the plot in the plot list you want to edit and then click "Edit Plot". The plot edit form will then appear. See Detail Control below for further help.

Special Multiple Plots on a Page

While users can add as many plots on a page as they wish, two special types of multiple plot graphs are predefined. The first is a 2D plot with projections. If from the graph definition form you pick a 2D plot and then click "2D Project →", three copies of the 2D plot are placed in the plot collection list. (Any pervious plots in the collection are removed.) The first plot will be a flat 2D, gray scale plot of the data in the upper left corner. The second plot is a x projection (sum over y) of the data. It is in the lower left corner. The third plot is a y projection (sum over x) of the data. It is in the upper right corner. The lower right corner is blank. The x projection plot has been resized and positioned so that its x-axis display length matches the x-axis on the 2D plot. Both the x and y projection plots have their x and y plot limits locked to the display limits on the 2D plot. This means that if you zoom in on an area in the 2D plot, the projection plots will follow the zoom.

The second type of special multiple plot is the overlay plot. If you select a histogram, and then "Overlay →", a plot will be added to the collection with all parts of the plot turned off except for the data line itself. This is, the title, axis labels and scales, tick and grid marks, etc. will be off. The only graphics the plot will generate is the data line. In the special condition that this new plot is the second plot in the collection and both plots are 1D plots, then the second plot will have its scales automatically locked to the first plot.

Graph Title Control

At the top of the Graph Definition form are controls for an optional page title. The big white box is a text box in which you can type in your title. For it to be visible, check the box “Show Title”. To change the font style, click on “Font …”. A font control dialog box will appear. You can change the color of the font either in the Font dialog box or by using “Text Color …”. The latter gives you a wider range of colors to chose. Finally, the background color for the page can be selected using “Background Color …”.

Keeping Changes to Graph and Plot Definitions.

Changes to plot and graph definitions are only kept if “OK” is clicked on the Graph Definition form. All changes to the graph and plots are discarded if “CANCEL” is clicked.

Edit Plot Control

To change the appearance of a plot, select it in the plot collection list of the Graph Definition form, then click “Edit Plot …”. The Edit Plot form will then appear. This has three major parts.

Caption. The caption of the form tells you which plot in the collection is currently being edited.

Appearance Controls. This is a set of tabs that have controls for defining various aspects of the plot.

Edit Plot Selection Controls. This is the set of buttons on the bottom of the form. With these buttons you can temporarily save plot changes and cycle through the plot collection.

Plot Appearance Controls

There are seven tabs that have controls that effect the appearance of a plot.

Placement. The Placement tab is used to control where a plot a appears and its size on the page. The big box on the right side of the control shows where on the page the plot will appear. For 1D plots, the plot itself is drawn, for 2D plots, only a representational box is shown for speed considerations. The units of the placement grid are defined by the controls “Number of Columns” and “Number of Rows”. Column and row counts start at 1. For example, if you want 4 plots in a 2 × 2 arrangement, you would set the number of row and columns to 2. Each plot would have a height and width of 1. Top and Left would be set to (1,1), (1,2), (2,1) and (2,2) for the four plots. On the other hand, if your are trying to establish some special effect, you might set the number of row and columns to a high number, thereby giving you finer positioning control. Note that each plot can have its own separate grid scale. The top of the placement gird is measured from the bottom of the page title if present. The individual controls on the form are:

Placement Display Box. This is the big picture on the right. It shows where on the page the plot will appear.

Plot Rotation. Allows the plot to be rotated on the page. Not yet implemented.

Top. The row number for the top of the plot.

Left. The column number for the left of the plot.

Height. Height of the plot in rows.

Width. Width of the plot in columns.

Number of Columns. Number of columns in the placement grid. Column numbering starts with 1.

Number of Rows. Number of rows in the placement grid. Row numbering starts with 1.

Update Display. Update the Placement Display Box. This does not save changes. You must click on “Save” at the bottom of the form to save any changes.

Force Aspect and Current Aspect. Allows the user to force the aspect ratio. Not implemented.

Display Only Data Range. If this box is checked, plot displays will always be limited to the defined region of the histogram. Plot limits outside this region will be ignored. If not checked, the user supplied plotting limits will be used without regard to the histogram limits. Histograms have a default value of zero outside their defined limits.

Auto Place. This brings up a new form that allows you to define how to arrange multiple plots on a page.

- The meanings of the values you enter are:

  Number of Plots Across: number of columns in placement matrix

  Number of Plots Down: number of rows in placement matrix

  Assignment order. Does the program go across or down first in assigning spots in the placement matrix.

  Start column: Column (1 to n) for the first plot.

  Start row: Row (1 to m) for the first plot.

  Step Column: step increment when increasing column number

  Step Row: step increment when increasing row number.

  Restart assignment after every xx plots. Used when plot overlays are in the list of plots to place. Resets the plot placement to the value given in start column and start row after every xx plots.

- How to use (no overlays):

  Starting with a blank graph definition, add plots to the plot list using the "ADD ->" button. Then go to Edit Plot. Click on "Auto Place". The program will come up with a tentative assignment scheme. Modify as you wish, then click OK. The plots will now appear in their new locations. If you want to add plots after the placement assignment has been done, you need to look at the values in the placement form closely, as they may look a little odd. It might be best to start with a new graph.

- How to use with overlays:

  Multiple overlays can now be handle easily using this form. As an example, say you have two plots, each with three different cuts. You want to display the two plots, with the different cuts overlaid. Do the following:

  From a blank plot list for the graph, add plots in this order:

  "ADD ->" Plot 1, cut 1

  "ADD ->" Plot 2, cut 1

  "Overlay ->" Plot 1, cut 2

  "Overlay ->" Plot 2, cut 2

  "Overlay ->" Plot 1, cut 3

  "Overlay ->" Plot 2, cut 3

  Then go to "Edit plots." Then click on "Auto Place". You will see that the program is suggesting a 1x2-placement matrix with reassignment restarting every 2 plots. That is, it detected the overlays and adjusted the placement accordingly. Click "OK". The program will then ask you if you want to lock the overlaid plots to the first plot of each type. Click "Yes". Plot placement will now occur and overlay plots will have their scales locked together. The overlay plots will also be assigned different line colors. Its that simple.

Style. The style section defines the general form of the plot. Here is where you may manually define plot limits, scale types, and data presentation style.

Plot Limits. This section is used to define the plot limits for the display and whether the scale is linear or logarithmic. Note the controls for the first two limits are labeled Horizontal and Vertical, not X and Y. That is because for 2D projections, this is a clearer designation of the axis being controlled. If the minimum and maximum limit are set equal (usually zero), the program will use the plot's default values. The default values for an axis are the histogram limits and 110% of the highest bin value for 1D plots or projections.

Scale Locking. The limits section is where you may lock a plot's axis to another plot. First, pick which plot you want to lock to by clicking "Change" next to the "Lock to Plot" label. A list of plots in the graph will appear. Pick you master plot. Then pick which axis you want to lock. If an axis is locked to another plot, a user's changes to the plot limits for that axis will be ignored.

2D Control. This section tells how 2D plots will be displayed. Choices are X Projection (y sum), Y Projection (x sum), or a Flat view with a color scale. Other 3D views are not yet defined. For the Flat view, "Scale" says that a color scale legend should be drawn on the right side of the graph. "B/W" distinguishes between a black and white scale or a color rainbow scale. "Invert" allows you to invert the gray or color scale.

Connect the Dots. This sections tell how the data points are to be connected for 1D and projection plots. You can use a step line, a straight line, or symbols. Any combination is allowed. If symbols are used, you may specify the interval for showing the data. For example, a spacing of 3 means every third data point has a symbol displayed. If a symbol is desired, use the symbol selection area to select the symbol desired. The "Font …" command currently does not affect anything.

Fonts. The Font tab controls how and whether axis labels and scales appear. For now, the Font… command can only control the color of the text. Later program versions will allow true font choices to be made. Similarly, the Format sections are currently inoperative. What you can control is color and whether an object is shown or not. So most of this control should be considered "Under Construction."

Lines and Fill. This tab controls the outline and fill of various regions and the histogram line type. The three regions are the plot area, the smaller area within the plot that data is plotted within, and the z axis color scale area. The boxes on the left edge of the control show the current look of the fill region. On the bottom of the control is the appearance of the histogram data line. You can change the appearance of this line by clicking "Change …". This will bring up the line format dialog box. Here you can change the line's color, thickness, and style.

Grid. The Grid tab controls the appearance of grid lines. A major grid line is drawn at the position of the major tick marks. Minor grid lines are drawn at the location of minor tick marks. The Zero grid lines are drawn on the axis. User 1 and User 2 are the optional lines defined by the /xd and /yd options in histogram definition script. The user optional lines are not yet implemented. The check boxes in upper part of the form controls the visibility of the lines (i.e., whether or not they are drawn). The "Change …" buttons at the bottom of the form allow you change color, line style, and thickness of the appropriate grid line.

Ticks. The Tick tab controls which ticks are shown and their interval. The first part of the form controls ticks visibility. Tick color and thickness is controlled by the "Change …" button on the bottom of the form. The tick spacing controls have yet not been implemented.

View. The view tab shows you what the current plot looks like by itself.

Edit Plot Selection Controls

There are five buttons at the bottom of Edit Plot form:

OK. Temporarily saves any outstanding changes and goes back to the Graph Definition form. Changes do not become permanent until the Graph is saved by clicking "OK" on the Graph Definition form.

Save. Temporarily saves any outstanding changes to a plot. Changes do not become permanent until the Graph is saved by clicking "OK" on the Graph Definition form.

Cancel. Remove any changes back to the last Save. The program then goes back to the Graph Definition form. If you want to cancel changes without going back to the Graph Definition form, click on "Next Plot". Answer No to the question about saving changes. Then cycle around to the original plot.

Next Plot. Cycle through the plot collection to the next plot. If changes have been made to the current plot and not saved, the program will ask if the changes should be saved.

Help. Someday this may do something.

Zoom

Zooming comes in two forms. The regular zoom command is used to select an area on the plot for expansion. The Zoom Z command is used to change the Z-axis scale on 2D plots. To go into zoom mode, select either Zoom or Zoom Z under the Plot menu. You will get a crosshair cursor in the active graph window. Use the mouse to move the cursor. The coordinates of the cursor are shown at the bottom, left corner of the plot. To define the expansion region, move the mouse to one corner of the new area, and then push the left mouse button. Keeping the left button down, drag the cursor to the opposite corner of your new box, then lift up on the button. The plot will then be redrawn. Normally, if your new box goes outside the defined region of the histogram, the plot limits will be reset to the defined limits of the histogram. This can be turned off in the plot edit control.

If there are multiple plots on the graph page, you must pick which plot you want to zoom on. Use the menu combination Plot / Pick Zoom/Gate… to bring up a list of plots in the current window. Pick which plot you want to zoom on. When you do the zoom, the labels giving the cursor position will appear in the lower left of the selected plot.

If other plots on the page have their scales locked to the plot you are zooming on, their scales will be changed appropriately. If the plot you are zooming on is already lock to another plot, your changes to the locked scale will be ignored.

While you are "zooming," no data is being taken. You can stop by clicking the right mouse button. Otherwise, the zoom feature will turn itself off in 30 seconds.

Printing

First make the window you want to save the active window by clicking anywhere on it. Then use the menu combination Edit / Print (or Control-P). The output goes to the currently defined default printer. To change the printer, change the default printer in the Print Manager. Likewise, to change printer properties such as orientation, change the printer properties using the Print Manager. The default condition for printer plots is to append the text summary to the page. If you don't want this, click on "Append Notes."

Update, Reset, and Redraw

You can cause the currently active plot window to be redrawn in three modes. The commands are under the Plot menu. Redraw draws the plot(s) with the current values of the histogram, but with all of the scale limits unchanged. Update draws the plot(s), but updates the data axis to match the current maximum value. Reset draws the plot(s) with all of the scale limits reset to their default values. Commands are provided to allow you to replot all open windows at the same time. Finally, shortcut keystrokes are provided for all the replotting commands (see the Plot menu for the keystrokes).

Text summary

With each plot, there is a short text summary available that lists such information as boundaries and averages. You can view the text by toggling the "Text" item in the Plot menu. Averages are reported only for the portion of the plot that is currently visible. Underflows and overflows give the counts outside the defined histogram limits, regardless of what the visible plot limits are.

One Dimensional example:

```
[COMMENT]
 Run Number: 460
 Data Time: 7/2/96 10:19:31 PM
 Clock Time: 8/26/96 7:36:19 PM
------------------------
Plot: S1 TDC
Event, Block, Index:  14,  1,  11
No test defined.
For Display Range Only
    FirstXBin, LastXBin:  1,  200
    Xmin, Xmax, Bin Size:  100,  300,  1
    Sum of Bins: =  29355
    X Average (bin centered): =  206.5001 ± 2.474714E-02
    X Sigma: =  4.240002
    Y Average: =  146.775 ± 3.509846
    Y Sigma: =  601.3525
For Entire Plot:
    Number of Underflows: =  0
    Number of Overflows: =  2
```

Two Dimensional example:

```
[COMMENT]
 Run Number: 460
 Data Time: 7/2/96 10:19:31 PM
 Clock Time: 8/26/96 7:37:01 PM
------------------------
Plot: Set 1, X vs theta raw
Event, Block, X Index, Y Index:  15,  2,  307,  216
Test:  51, Test Count:  8658
For Display Range Only
    FirstXBin, LastXBin:  1,  1000
    Xmin, Xmax, Bin Size: -10,  10,  .02
    FirstYBin, LastYBin:  1,  40
    Ymin, Ymax, Bin Size: -.03,  .03,  .0015
    Sum of Bins: =  8657
    X Average (bin centered): =  8.260252E-02 ± 4.563814E-02
    X Sigma: =  4.246309
    Y Average (bin centered): = -8.724151E-05 ± 9.881034E-05
    Y Sigma: =  9.193611E-03
    Z Average: =  .216425 ± 1.369031E-02
    Z Sigma: =  1.273788
For Entire Plot:
    Number of Overflows: =  1
```

Stepping Through Plots

You can step through plots using the Next Plot and Next Page commands under the plot menu. Shortcut key are defined (see the menu). For graphs with only one plot, the two commands are the same. Pressing Next Plot changes to the next plot in the internal histogram list with the same plot type. (Previous Plot and Previous Page go backward through the list.) If you have multiple plots, then there is a difference. For Next Plot, the histogram index is incremented by one. For Next Page, the histogram index is incremented by the number of plots on the page. You use Next Plot, for instance, if all the plots on the page are the same histogram viewed different way, like a 2D plot with projections. Use Next Page if you are looking at a series of plot. For example, if you are looking at hit frequency plots for chambers 1 to 4, then press Next

Page, you would get chambers 5 through 8. The histogram list order is defined by their appearance in the command script.

Histograms

See the section on script formats for histograms until this section is finished.

Time

1D

2D

Multiple entries

Starting Replay from an Existing HSV File

When you replay multiple runs, you often want to accumulate histograms over those runs. Sometimes it will be desirable to start the accumulation with a pre-existing HSV file. With care, this is easy to do.

Run your default setup script file.

Go to the Run / PC DAQ form and initialize the run.

Use the browse button to get the HSV file.

Click Start to run the HSV file. The HSV file is then read in.

Use Browse… button to get your new replay data file (*.dat) or batch file list (*.bat).

Under the Menu command sequence Options / Replay…, the replay command form will appear. TURN OFF "Histogram Data." This option was set by program when you earlier selected the HSV file. Failure to turn it off will wipe out the accumulating data.

Turn off "Clear Histograms at Run Start" if on.

Start the data replay.

Adding Histograms after the Fact

Occasionally you want to add histogram definitions and maybe data to the list already in memory without resetting or zeroing what is already present. This can be done in two ways. First, they can be added from a script file using the [AHI] section command instead of the [HIST] section command. AHI mean Append Histogram. All the same syntax rules apply. [AHI] simply skips over the reset code when it used.

The second way to append data is to read in a HSV file with the new histograms defined within it. After you pick the HSV file for reading, go to the menu Options / Replay… to get the replay control form. Turn OFF "Reset Histogram List." You will probably also want to turn off "Clear Histograms at Run Start" on the run control form. Then start the HSV run. Remember that the appended histograms should have unique names.

After you have created your expanded histogram list, it is probably a good idea to save this setup to a new HSV file, and then use the new expanded file as your HSV replay file.

Accessing Histogram Data in ANALnn Programs

Sometimes you will want to get at the contents of a histogram from within an ANALnn program. You would most probably want to do this in response to the "uEndOfRunCall" call. Note that this call is generated whenever a run ends, whether the run is live data, replay data, or an HSV file. All that is needed is to enable the event and user analysis for the event. Thus, the way to write an analysis of HSV data is to pick an unused event number and then place your analysis code in the End-of-Run section. Then add event and user enable commands to your default command script. Then, whenever you read in the HSV file, the analysis code is automatically called. Also, the begin-of-run analysis is done for enabled events, which would mean your database is read-in if it is one of the enabled events.

The following routines are provided to get at histogram information. A general histogram manipulation program set would have many more routines, but that must await future upgrades.

Integer*4 Function **GetHistIndex**(Name, EventNumber, BlockIndex, HistogramType)

```
Character*(*) Name          ! Name (title) of the histogram.
Integer*4 EventNumber       ! Event number of histogram
Integer*4 BlockIndex        ! System block for histogram data source
Integer*4 HistogramType     ! Histogram type (0=time, 1=1d, 2=2d)
```

This routine gets the index to the histogram list for the specified histogram. Only the first 40 characters of Name are used. If EventNumber or BlockIndex are zero, they are ignored. Remember that the combination of the these four quantities should uniquely specify a histogram. If the histogram does not exist, -1 is returned.

Subroutine **ClearHistogram**(Index)

```
Integer*2 Index
```

This routine clears (zeros) the specified histogram.

Integer*4 Function **Get1DPointer** (hIndex, XValue)

```
Integer*4 hIndex
Real*8 Xvalue
```

This program calculates the absolute location in the array HistBuffer for a given histogram and value. Thus, if you want to set the value for histogram 5 for x=3.4, do:

```
HistBuffer(Get1DPointer(5,3.4)) = value
```

Remember that HistBuffer is an I*4 array. Get1Dpointer properly checks for and adjusts its output if an underflow or overflow occurs. If the histogram is not defined, it returns a value of -1.

Integer*4 Function **Get2DPointer** (hIndex, XValue, YValue)

```
Integer*4 hIndex
Real*8 XValue
Real*8 YValue
```

This program calculates the absolute location in the array HistBuffer for a given histogram and value. Thus, if you want to set the value for histogram 5 for x=3.4 and y=9, do:

```
HistBuffer(Get2DPointer(5,3.4,9)) = value
```

Remember that HistBuffer is an I*4 array. Get2Dpointer properly checks for and adjusts its output if an underflow or overflow occurs. If the histogram is not defined, it returns a value of -1

Subroutine **HistRatio1D** (Numerator, Denominator, Result)

```
Integer*4 Numerator
Integer*4 Denominator
Integer*4 Result
```

This routine computes the ratio of two histograms and put the results in a third. If the denominator is zero, it divides by 1 instead.

Subroutine **HistRatio2D** (Numerator, Denominator, Result)

```
Integer*4 Numerator
```

```
Integer*4 Denominator
Integer*4 Result
```

This routine computes the ratio of two histograms and put the results in a third. If the denominator is zero, it divides by 1 instead.

Subroutine **HistStat2D** (hIndex, FirstXBin, LastXBin, FirstYBin, LastYBin, XArray, Yarray, SumZ, SumZZ, SumZX, SumZXX, SumZY, SumZYY, SumZXY, zMaximum, zMinimum)

```
C
C  Input parameters
C
      Integer*4 hIndex      ! Index number of Histogram
      Integer*4 FirstXBin   ! Bin number of first X bin (1->.xbins)
      Integer*4 LastXBin    ! Bin number of last X bin
      Integer*4 FirstYBin   ! Bin number of first Y bin (1->.xbins)
      Integer*4 LastYBin    ! Bin number of last Y bin
      Real*8    Xarray(FirstXBin:LastXBin)    ! Array of x values
      Real*8    Yarray(FirstYBin:LastYBin)    ! Array of y values
C
C  Output Parameters
C
      Real*8    SumZ
      Real*8    SumZZ
      Real*8    SumZX
      Real*8    SumZXX
      Real*8    SumZY
      Real*8    SumZYY
      Real*8    SumZXY
      Real*8    zMaximum
      Real*8    zMinimum
```

This routine computes various sums involving both the data in a bin and the coordinate of the bin.

Subroutine **HistSum2D** (hIndex, FirstXBin, LastXBin, FirstYBin, LastYBin, SumZ, SumZZ, zMaximum, zMinimum)

```
C
C  Input parameters
C
      Integer*4 hIndex      ! Index number of Histogram
      Integer*4 FirstXBin   ! Bin number of first X bin (1->.xbins)
      Integer*4 LastXBin    ! Bin number of last X bin
      Integer*4 FirstYBin   ! Bin number of first Y bin (1->.xbins)
      Integer*4 LastYBin    ! Bin number of last Y bin
C
C  Output Parameters
C
      Real*8    SumZ
      Real*8    SumZZ
      Real*8    zMaximum
      Real*8    zMinimum
```

This routine computes various sums involving only the data in a bin but not the coordinate of the bin.

To get at a histogram's data, you must first find the index to the histogram list. This is provided by the routine GetHistIndex. You should always check that the lookup was successful. Look at the code in HistRatio2D for an example of how to get and put data into the histogram buffer. Even if a histogram is to be computed rather than accumulated, its definition in the script file must still give block and index information even though it is meaningless. So make up something.

Changing Histogram Names and Axis Labels

It may happen that you need to change a histogram's name or axis label without rerunning a script. This usually happens because you already have data in the histogram or the histogram comes from an HSV file. You can do this in the Histogram definition control form. Use the Menu combination Options / Histograms… to get to the form. Make the changes you want, then click "Change" to save the changes. If you are changing the name in a HSV file, you should save the histograms to a new histogram file.

Saving Multiple Histograms as a Binary File

You can manually save histograms to a HSV file at any time. At the bottom of the run control form is a button "Save Histograms." Clicking this button brings up the Histogram Save form. You can save (1) all histogram, (2) only those histograms that have the /lo flag set in their script definition, or (3) a selection you will pick for the list on the right of the form. The right hand list responds to the standard Windows list selection commands. Once your selection is done, click on "Save Histograms" to get the file dialog box. Other ways of saving histogram information are in the section on saving plots.

Data Screen

This section has not been written yet

Q Test Package

GENERAL CONCEPTS

Purpose of the Test Package

The purpose of the test package is to provide the experimenter or programmer with a dynamic, convenient means of examining and classifying data on an event-by-event basis. Data of that kind would typically arise either in a data acquisition/replay environment or in a Monte Carlo style calculation. The package is convenient in that only a minimal amount of information need be specified in the program at the time it is written. It is dynamic in that all details about what tests are to be performed are specified in a user-created file called the Test Script. This way of doing things means that the tests may be changed without the need to compile and link the program.

Overview of Capabilities

The package is quite powerful in that any set of tests expressible in LOGICAL or ARITHMETIC IF statements in FORTRAN may also be performed within the framework of the test package.

There are three basic types of tests: ARITHMETIC, USER, and LOGICAL. The ARITHMETIC test is used to examine the values of data words for such purposes as setting gates on values, checking for equality, and examining bit patterns. The USER test is provided so that users may incorporate the logical results (.TRUE./.FALSE.) of their own "tests" into the framework of the test package. The LOGICAL tests are used to form various logical combinations (.AND., .OR. (inclusive and exclusive), and .NOT.) of the results of any of the previously performed tests.

The tests are performed together in groups called BLOCKS. All tests within a given BLOCK are executed in response to a single subroutine call (TSTEXE) during the analysis of an event. The analysis subroutine for an event may contain only a single BLOCK of tests, or it may have several. After execution of a BLOCK of tests, the results (.TRUE. or .FALSE.) are stored in Test Result Flags and are available within the analysis routine for any use desired. The BLOCKS used need not be the system blocks. Possible uses would be to control the program flow or to control entry into a user-written histogramming routine. Also included for each test, is an I*4 counter which is incremented each time that test is passed. A similar counter exists for each BLOCK that keeps count of the number of times that block of tests is executed. These counters are useful for summary information (e.g., what fraction of all events hit counter #3). The values of these counters are available both within the program and also to separate programs.

Users should be aware that the test block numbers do not have any necessary correlation to the block numbers defined for the histogramming system. However, as a matter of convenience, it may be very useful to make the block number for a set of histograms the same as the block number for the tests which are used to control entry into those histograms.

You can view the list of defined tests by looking at the Test Definition control. You bring this up by the following menu commands: Options / Tests… / . This control form has three drop down lists that show the current set of tests, gates, and boxes. At the moment, this control is used to display data only. No editing can be done. You may, however, generate a script based on the current list of tests by clicking on "Generate Script." This will dump the current set of test definitions into the "Replay Script" window. This can be used to generate text script commands for indirect gates that have the current values as their default setting.

You can view and control the current value of the test counters by looking at the Test Screen. You bring this up by the following menu commands: View / Test Screen /. This brings up the Test Screen Control. You can make the viewing area larger by resizing the form or you can scroll the list box. This control allows you to reset the counters, print them, or copy them to the clipboard (for pasting into, say, a spreadsheet).

Room has been allocated for 500 tests in 100 blocks and including 100 Indirect Boxes and 100 Indirect Gates. This is set by the parameters: NumberOfTests, NumberOfTestBlocks, NumberOfTestGates, and NumberOfTestBoxes in both the FORTRAN and VB code.

Connection to Event Numbers

The test package is intended for use both in the data acquisition/replay environment and in Monte Carlo calculations that may have quite a different structure. For this reason, no connection is made directly between tests and event numbers. However, the breakup of tests into BLOCKS lends itself nicely to use within the PC DAQ framework.

If the analyzer has only one event type on which tests are to be performed, there is no problem. The block number(s) of the tests can be chosen as desired and need not bear any relationship to the event number. In the case that more than one event type is making use of the tests, one must decide at the time the program is written which test BLOCKS will be used in which event processor. Since all the tests in a BLOCK are performed together, it is not possible to have a single BLOCK serve for two events. It is possible for the LOGICAL TESTS in one event number to make use of the results from the tests in a different event number, but such applications do require careful programming and are not recommended for the novice.

Connection to Histogramming

Although the Test Package may be used separately from the PC DAQ standard histogramming subsystem, a number of benefits accrue if it is used in conjunction with it and its associated support programs. The first is that the results of the tests may be used to control whether or not a given histogram is incremented. This ability comes essentially for free and can be specified by the user when the histogram is defined. The second benefit is that the plotting system can be used with the mouse to specify data word indices and limits for certain of the ARITHMETIC TESTS called INDIRECT GATES and INDIRECT BOXES. The third benefit is that the entire test data base, including the values of the test and block counters, and the indices and limits of the indirect gates and boxes, can be saved.

User Information

This section is intended to provide information necessary to operate the test package, assuming that the programming has already been done. This section will provide the information needed to write a test descriptor file and to set up and modify the data limits for the INDIRECT GATES and BOXES.

A certain amount of cooperation is necessary between the programmer and the person who writes the test descriptor file if the test package is to be useful. It is necessary to coordinate which BLOCK numbers will be used in which location since these must be specified both in the analysis subroutine and in the Test Descriptor File. It is also necessary to have some idea of where in the course of analysis each block of tests is being executed. It makes little sense to test a value which has not yet been calculated for a given event or which is not contained in the array specified in the call to TSTEXE. Beyond these relatively simple considerations, the person who writes the test descriptor file is pretty much able to operate independently of the programmer.

The test descriptor file is the mechanism for specifying nearly all information about what tests are to be performed on the data. The only exception to this rule is that the limits for the so-called INDIRECT GATES and INDIRECT BOXES.

There are three types of lines allowed in a test descriptor file:

- Comment lines
- Block delimiter lines
- Test descriptor lines

Each will be discussed in some detail below and a sample Test File.

There is no particular order in which the blocks must be defined. A given block of tests is referenced at execution time only by the block number specified in the call to TSTEXE. This means that the order of

execution of various blocks does not depend on the numerical sequence of block numbers or the order in which they were defined in the Test File.

An error will result if the delimiter for the specified BLOCK has previously been encountered in the file.

Users should be aware that the setup program has no way of knowing whether you are defining tests for a block which will never be called in the program executing the tests or whether that program will call a block for which no tests have been defined. Neither of these situations can result in an error condition in compiling a script. In the former case, an examination of the Test Screen will indicate that the block was never executed. In the latter case, an error condition code will be returned by the subroutine (TSTEXE) in the analyzer, but no error message will be printed.

It should also be noted that it is perfectly legal to have a BLOCK containing no tests. This would occur if the file contained two successive block delimiter lines. In this case, since the BLOCK has been defined, no run time error would occur and only the block counter would be incremented each time the appropriate call was made to TSTEXE.

Test Descriptor Lines

These lines are used to specify the tests to be performed. One and only one test can be specified per line. The general format of each line is that of a test number followed by the appropriate test description. The syntax for each test along with its description will be given in Test Scripts. The test numbers should be considered as numeric labels for tests. They have nothing to do with the order in which the tests are performed. That is determined by their order as encountered in the file and the order in which the specific blocks are executed by the user program.

A given block of tests will be considered ended when the next block delimiter line or the end of the file is encountered. Within a given block of tests, the number you may assign to each test is arbitrary as long as three conditions are met. First, it must not be larger than the value of NumberOfTests parameter given in the PCDAQ global include files (VB and FORTRAN). Secondly, it must not duplicate an already used test number. Unlike the Q test package, different blocks may have overlapping test numbers. For example, if BLOCK 1 has tests ranging in number from 1 - 40, other block may contain a test number in that range. If any of these restrictions is violated, an appropriate error message will be issued by PC DAQ when you run the script.

Test Helpful Hints

While it is not required, it is probably a good practice to place all LOGICAL TESTS in a given block after all ARITHMETIC TESTS in that block. This lessens the likelihood that a LOGICAL TEST might be used to refer to an ARITHMETIC TEST that has not yet been performed. It is also probably good practice to number tests within a given BLOCK sequentially when the TEST DESCRIPTOR FILE is first being written and to leave a gap in the test numbers between BLOCKS. This allows for tests to be inserted later in any BLOCK without having to worry about test numbers overlapping for different BLOCKS.

Description of Tests

This section will describe in detail the function of each of the tests and the syntax for the descriptor line in the TEST DESCRIPTOR FILE. A few general rules applying to all test descriptors are given below:

- Unless otherwise specified, all test numbers are integers. Test parameters are saved as double precision numbers.
- Within a test descriptor, commas, spaces, and tabs will be taken to be delimiters between various parameters of the test. A succession of tabs or spaces will be treated as a single tab or space, but successive commas will be taken as separate delimiters. This allows for tabs and spaces to be used freely to create a neater file format.
- Each test descriptor must begin with a user-assigned test number. This value (NTST) must fulfill the condition $0 < |NTST| \leq$ NumberOfTests. The value of NTST must be followed by a delimiter. If NTST is negative, then the specified test must fail for the test to be true.

- The second element on each descriptor must be the type for the desired test. It is always sufficient to enter only the first two characters of the type, although more may be typed for intelligibility. The test type must be followed by a delimiter.

Following the test name are the parameters (if any) of the test. All parameters are separated from one another by delimiters.

All arithmetic tests specify a data word index for the data word to be examined. This index is the same as that one would specify for an array in FORTRAN, with the first index in the array being 1 not 0. A data word index with a value < 1 will be flagged as an error. Exactly which array is being referred to is determined by the call to subroutine TSTEXE. This is illustrated in the following example:

CALL TSTEXE (NBLK, IDAT,IERR)

Then a Data Word Index of 13 refers to IDAT(13) but

CALL TSTEXE(NBLK,JDAT(50),IERR)

Then a Data Word Index of 13 refers to JDAT(62)

The call to TSTEXE (the test call) has been upgraded so that the data array can be any type (I*2, I*4, R*4, or R*8). All data word indices in a given BLOCK of tests will be taken as referring to the same array. If it is necessary to perform tests on more than one array of data, such tests must be done in separate BLOCKS. In the case of INDIRECT GATE and BOX tests, the data word indices need not be specified in the test descriptor file. They are most commonly taken from information provided from the display using the graphic input cursor. However, they still refer to the array given in the call to the subroutine TSTEXE. In all examples used below, it is assumed that the call to the test execution subroutine is:

CALL TSTEXE(NBLK,IDWD(1),IERR).

In each logical test, you may specify both a set of tests you want to have been passed (i.e., be .TRUE.) and a set of tests you want to have been failed (i.e., be .FALSE.). In all Logical tests, you refer to the tests you want to be .TRUE. by their test number, and to the tests you want to be .FALSE. by their test number preceded by a minus sign (-). Users should take particular note that a test which has not yet been performed in the analysis of the current event is taken to be .FALSE.

BIT TEST

NTEST, BI (t), INDX, IBITNO

The BIT TEST looks to see if the specified bit in a given data word is set to 1. If it is set to 1, the test is .TRUE.; otherwise, it is .FALSE. The status of any other bits in the data word is irrelevant. The test is .FALSE. if a floating point data array is used in the call to TSTEXE.

Example:

23,BIT,15,1

Test 23 is .TRUE. if IDWD(15) = 6(octal)

Test 23 is .FALSE. if IDWD(15) = 5(octal)

EQUALITY TEST

NTEST,EQ(ual),INDX,VAL

The EQUALITY TEST is .TRUE. if the specified data word has exactly the value given. NTEST, INDX, are integer values. VAL is double precision.

Example:

2,EQ,120,-352

Test 2 is .TRUE. if IDWD(120)=-352.

PATTERN TEST

NTEST,PA(ttern),INDEX,MASK,IVALUE

The PATTERN TEST checks to see if the given data word, after masking for bits of interest, is identical to the specified value on a bit for bit basis. For convenience, MASK and IVALUE are expressed as OCTAL values (O8 format). The test is .FALSE. if a floating point data array is used in the call to TSTEXE. The test is .TRUE. if:

IAND(MASK,IDWD(INDEX)) = IVALUE.

Example:

25,PATTERN,29,176,170

Test 25 is .TRUE. if IDWD(29) = 30171(octal) {IAND(176(octal), 30171(octal)) = 170(octal)}

Test 25 is .FALSE. if IDWD(29) = 173(octal) {IAND(176(octal), 173(octal)) = 172(octal)}

GATE TEST

NTEST,GA(te),INDEX,LIMLO,LIMHI

The GATE TEST checks if the specified data word is both ≥ some lower limit (LIMLO) and < some upper limit (LIMHI). Thus, the test is true if:

(IDWD(INDEX) ≥ LIMLO) .AND. (IDWD(INDEX) < LIMHI)

Example:

49,GATE,33,-250,100

Test 49 is .TRUE. if (IDWD(33)≥-250).AND. (IDWD(33)<100).

INDIRECT GATE TEST REFERENCE

NTEST,IG(ate),NGATE

The INDIRECT GATE TEST operates exactly the same as the normal GATE TEST except that the data word index and the upper and lower limits are provided indirectly. They may either be taken from a plot of a histogram or they may be entered manually using the /GA control. The first mode would be the most common method in actual use. The latter mode would likely be used only if one needed to set the limits to some previously determined values. Abs(NGATE) must be a number between 1 and NumberOfTestGates.

NOTE: When referring to an INDIRECT GATE for purposes of setting up data word indices and limits one ALWAYS uses the INDIRECT GATE NUMBER and NEVER the test number. If no values were previously defined for a given indirect gate or box, then all data word indices and upper and lower limits will be set to zero. In particular, this means that such tests will never be true until these values have been set up.

Example:

99,IGATE,2

Test 99 consists of INDIRECT GATE No. 2.

INDIRECT BOX TEST REFERENCE

NTEST,IB(ox),NBOX

The INDIRECT BOX TEST is basically a combination of two INDIRECT GATE TESTS. Its main use is to define a rectangular region of interest on a two-dimensional histogram. Abs(NBOX) must be a number

between 1 and NumberOfTestBoxes. It requires two data word indices (INDX1,INDX2) and two sets of lower (LIMLO1, LIMLO2) and upper (LIMHI1, LIMHI2) limits. These data words either may be given explicitly using the /BO command or may be taken from a two-dimensional histogram plot. This test is true if:

(IDWD(INDX1) ≥ LIMLO1) .AND. (IDWD(INDX1) < LIMHI1) .AND. (IDWD(INDX2) ≥ LIMLO2) .AND. (IDWD(INDX2) < LIMHI2)

Example:

15,IBOX,4

Test 15 consists of BOX No. 4.

AND TEST

NTEST,AN(d),N1,N2[,...,Nm]

The AND TEST looks to see if all specified tests are in their requested state (.TRUE. or .FALSE.). Tests one desires to be .TRUE. are referred to by their test numbers. Tests one desires to be .FALSE. are referred to by their test numbers preceded by a minus sign (-). No specific order is required in specifying the test numbers.

Example:

15,AND,2,27,-3,5

Test 15 is .TRUE. if Tests 2, 5, and 27 are .TRUE., and Test 3 is .FALSE.

INCLUSIVE OR TEST

NTEST,IO(r),N1,N2,...Nm

The INCLUSIVE OR TEST is .TRUE. if any of the specified tests are in their requested state (.TRUE. or .FALSE.). Tests one desires to be .TRUE. are referred to by their test numbers. Tests one desires to be .FALSE. are referred to by their test numbers preceded by a minus sign (-). No specific order is required in specifying the test numbers. The number of entries is limited to 10.

Example:

27,IOR,-2,5,-3,14

Test 27 is .TRUE. if either

Test 5 or 14 is .TRUE. or

Test 2 or 3 is .FALSE.

EXCLUSIVE OR TEST

NTEST,EO(r),N1,N2,...Nm

The EXCLUSIVE OR TEST is .TRUE. if exactly one of the tests specified as .TRUE. is .TRUE. or (exclusive) exactly one of the tests specified as .FALSE. is .FALSE. Tests one desires to be .TRUE. are referred to by their test numbers. Tests one desires to be .FALSE. are referred to by their test numbers preceded by a minus sign (-). No specific order is required in specifying the test numbers. The number of entries is limited to 10.

Example:

11,EOR,2,3,-5

Test 11 is .TRUE. if Test 2 or (exclusive) Test 3 is .TRUE. or (exclusive) Test 5 is .FALSE.

MAJORITY TEST

NTEST,MA(jority),NCNT,N1,N2,...Nm

The MAJORITY TEST operates much like a NIM majority logic unit. The test is .TRUE. if at least NCNT of all tests specified are in the desired state. Tests one desires to be .TRUE. are referred to by their test numbers. Tests one desires to be .FALSE. are referred to by their test numbers preceded by a minus sign (-). No specific order is required in specifying the test numbers. The number of entries is limited to 10.

Example:

15,MAJORITY,3,2,4,-6,-7

Test 15 is .TRUE. if at least three of the following occur: Tests 2 or 4 are .TRUE. or Tests 6 or 7 are .FALSE.

NOTE: Since the syntax for the MAJORITY TEST is slightly different than that of the other Logical tests, a warning message will be issued if the value of NCNT appears to be unreasonable (i.e., either £ 0 or > number of tests specified).

USER TEST

NTEST,US(er)

The USER TEST provides the facility for incorporating into the test package information not conveniently obtainable from the standard tests. Its use requires a close cooperation between the programmer and the person writing the TEST FILE, since the only information specified in the TEST FILE is what the test number is. The programmer MUST call a subroutine (TSTSET) in the analyzer to set the value of this type of test to .TRUE. It should also be noted that the test result flag for this test does not become .TRUE. nor is the counter incremented when TSTSET is called. Both those operations occur only when TSTEXE is called for the BLOCK containing that USER TEST.

Example:

15,USER

Test 15 is defined as a USER TEST.

INDIRECT BOX TEST DEFINITION

/BO:boxnum:[xindx:xlo:xhi:yindx:ylo:yhi]

This switch is used to enter the limits for Indirect Box "boxnum" manually. The values are self explanatory.

INDIRECT GATE TEST DEFINITION

/GA:gatnum:[indx:lolim:hilim]

This switch is used to enter the limits for the gate "gatnum" manually. The values required are self explanatory.

Sample Test Descriptor File

```
;
; The following is a sample Test Descriptor File
; for EXPT. 6.33 studying polarized pion scattering from
; Linoleum-223, Feb. 30-31, 1982 BC.
;
; Room has been allocated for 500 tests in 100 blocks and including
; 100 Indirect Boxes and 100 Indirect Gates.
;
```

```
BLOCK,1            !This begins first BLOCK of tests
1,GATE,122,0,100   !Gate Test on Data word 122
2,GATE,123,-100,0  !Gate Test on word 123
3,IGATE,5          !Indirect Gate #5,
20,IBOX,2          !Box #2, Test Numbers need not be ordered
;
; Indirect Gates and Boxes do not need to be defined in order
; nor do all Indirect Gates and Boxes for which room has been
; reserved need to be defined.
;
4,PATT,2,377,3     !Low byte of data word 2 must = 3
6,AND,1,2          !Tests 1 and 2 must both be .TRUE.
;Tests do not need to be sequential
9,EOR,3,4          !Either tests 3 or 4 must be .TRUE.
10,USER            !User results included by this test
11,AND,1,-1  !This test will never be .TRUE.
12,IO,1,-1         !This test will always be .TRUE.
13,MAJOR,3,1,2,3,4,6,9 !This test will be .TRUE. if at least 3 of
;                       ! Tests 1,2,3,4,6,9 are .TRUE.
;
; Here ends first block and begins block 3.
; Note, since block 1 contains tests ranging in number from
; 1-20, that other blocks may not have any test in this range. ;
;
BL,3               !Block 2 doesn't need to exist
;
;
21,IOR,23,24       !This test will never be .TRUE. since we have
;                  !not defined tests 23 and 24 yet
;
; This is the end of the tests.
; Not all 100 tests for which room has been allocated need
; to be defined.
;
; Indirect boxes and gates can also be defined in the script.
; Box and gate definitions can appear anywhere in the script.
; They are not associated with any block, regardless of where they appear
; in the script.
/bo:5:18   ! indirect box 5, index is 18
/ga:45     ! indirect gate 45
```

PROGRAMMER INFORMATION

This section is intended to provide the programmer with all the information needed to incorporate the test package into an analyzer or other user program. It will include sections describing each subroutine and function available to the user.

Subroutines and Functions

Note for old Q programmers, the subroutines TSTMAP and TSTINI are included only for compatibility with old code. They do nothing.

Subroutines are provided to clear test result flags, clear test and block counters, set the value of User tests, execute blocks of tests, and retrieve the current values of the test result flags and test and block counters.

As a general rule, all subroutines return an error flag to indicate the nature of any errors encountered. However, they do not issue any error messages on the user terminal. If the programmer desires that an error message be issued, such capability must be explicitly incorporated into the user program.

The following sections contain, in alphabetical order, the descriptions for all user subroutines and functions. A logical ordering and example of how they are likely to be used can be found elsewhere. The routines BCTDSK, DGTPRM, and TCTDSK do not exist in this implementation of the test package.

Retrieval of Block Counters - Subroutine BCTCOR

CALL BCTCOR (FIRST, LAST, COUNTS, IERR)

FIRST = Lowest block number (I*4-input)

LAST = Highest block number (I*4-input)

COUNTS = Block counters (I*4 array-output)

IERR = Error code (2 word I*4 array-output)

Error codes given as IERR (1), IERR (2) below

= 1,2049 success no errors

= -2,2049 illegal block number

This subroutine can be used to retrieve the (I*4) counters for one or several blocks of tests. The possible error conditions will be that a specified block number is unreasonable (< 1 or > NumberOfTestBlocks or undefined).

Retrieval of Test Counters - Subroutine TCTCOR

CALL TCTCOR (FIRST, LAST, COUNT, IERR)

FIRST = Lowest test number to be gotten (I*4 - input)

LAST = Highest test number to be gotten (I*4 - input)

COUNTS = Test counters (I*4 array-output)

IERR = Error code (2 word I*4 array-output)

Error codes given as IERR (1), IERR (2) below

= 1,2049 success no errors

= -2,2049 illegal test number

This subroutine can be used to retrieve the values (I*4) of one or several test counters. The possible error conditions will be that a specified test number is unreasonable (< 1 or > NumberOfTests). At this time, the subroutine does not check to see whether the specified test was actually defined in the Test Descriptor File.

Retrieve Test Parameters - Subroutine TGTPRM

CALL TGTPRM (IARRAY, IERR)

IARRAY = 24 element array (I*4-output)

IARRAY (4) = NumberOfTests

IARRAY (5) = NumberOfTestBlocks

IARRAY (6) = NumberOfTestGates

IARRAY (7) = NumberOfTestBoxes

Other entries are zero.

IERR = Error code (2 word I*4 array-output)

Error codes given as IERR (1), IERR (2) below

= 1,2049 success no errors

This array includes such information as the maximum number of tests, blocks, etc. It is probably most useful for determining the maximum number of tests and blocks for purposes of setting DO LOOP limits for retrieving counters.

Clearing Test Result Flags - Subroutines TSCLFB & TSCLFA

CALL TSCLFB (NBLOCK, IERR)

CALL TSCLFA (IERR)

NBLOCK = the block number (I*4-input)

IERR = Error code (2 word I*4 array-output)

Error codes given as IERR (1), IERR (2) below

= 1,2049 success no errors

= -2,2049 illegal block number

= -4,2049 Test not defined

Prior to executing a given BLOCK of tests for each event, it is necessary to set the Test Result Flags for those tests to .FALSE. The reason this is done in a separate subroutine rather than as each test is executed is to provide the capability of skipping certain blocks of tests while maintaining the condition that tests not yet performed for a given event are .FALSE.

The subroutines will allow the user to specify either a specific block number for which the test result flags are to be zeroed (TSCLFB), or he may specify that the flags for all blocks are to be zeroed (TSCLFA). Since it is substantially faster, the latter mode will likely be used in all experiments except those which are passing information from one event processor to another.

The only error conditions will be that the specified block or test has not been defined. These conditions will return error codes to the calling routine. No error message will be printed.

Test Execution Subroutine TSTEXE

CALL TSTEXE (NBLOCK, IARUSR, IERR)

NBLOCK = block number of tests (I*4-input)

IARUSR = array of data words to be tested (I*2, I*4, R*4, or R*8 input)

IERR = Error code (2 word I*4 array-output)

Error codes given as IERR (1), IERR(2) below

= 1,2049 success no errors

= -2,2049 illegal block number

= -4,2049 database is garbage

= -5,2049 bad indirect gate or box

All tests are performed in groups called BLOCKS. Each block of tests will be performed in response to a single call to the subroutine TSTEXE. The only error checking performed by TSTEXE prior to execution of the tests will be that the specified block has been defined. Error codes will be returned for each of these conditions, but no error message will be issued. It will be up to the user to issue an error message in this case. One fatal and, unfortunately, undetectable run time error is possible. This would occur if the Test Descriptor File specified an unreasonably large index for a data word to be tested via an Arithmetic test. In such a case, it is possible that the implied data word would lie outside the program address space and attempting to access it would cause a fatal memory access violation. Unfortunately, it is not possible to protect against such errors. An attempt to prevent this is made by checking the index against the parameter qMaxIndex, which is currently set to 100,000.

A potential source of difficulty is that no check is made as to whether or not a given block of tests is executed more than once for a given event. If this happens, counters for all tests which are .TRUE. will be incremented each time, and the test result flags will contain the logical .OR. of both calls unless TSCLFA or TSCLFB is called before each call to TSTEXE. Since executing a given block of tests more than once for a given event produces inconsistent test results, the procedure is strongly discouraged.

To have a common call for TSTEXE independent for block variable type, there are actually four versions of TSTEXE. They are TstExeI2, TstExeI4, TstExeR4, and TstExeR8. The interface statement for TSTEXE defines an overloaded interface that depends on the block variable type.

Subroutine TSTINI

CALL TSTINI (IERR)

IERR = Error code (2 word I*4 array-output)

Error codes given as IERR (1), IERR (2) below

= 1,2049 success no errors

Subroutine TSTINI is a null program.

Subroutine TSTMAP

CALL TSTMAP (IERR)

IERR = Error code (2 word I*4 array-output)

Error codes given as IERR (1), IERR (2) below

= 1,2049 success no errors

Subroutine TSTMAP is a null program.

User Test Control - Subroutine TSTSET

CALL TSTSET (NTEST, LVALUE, IERR) (internal only)

NTEST = Test Number (I*4-input)

LVALUE = Logical value (L*1-input)

IERR = Error code (2 word I*4 array-output)

Error codes given as IERR (1), IERR (2) below

= 1,2049 success no errors

= -2,2049 illegal test number

In order to accommodate the concept of a "User Test" into the test package, it is necessary to allow the user to specify the status of such a test. This is done through the subroutine TSTSET. It should be noted that TSTSET does not immediately cause the value of the appropriate Test Result Flag to be set to the value indicated nor does it increment the Test Counter. These two operations only happen at the time TSTEXE is executed for the BLOCK containing that USER TEST. It should also be noted that calling TSCLFA or TSCLFB for the BLOCK containing a USER TEST will set that Test Result Flag to .FALSE. invalidating any previous calls to TSTSET. TSTSET will have no effect on the Test Result Flag value unless that test has been defined as a USER TEST in the Test Descriptor File. Thus, an understanding must be reached between the programmer and the person who writes the Test File as to what test numbers will be USER TESTS.

The only error conditions will be that the test number specified was unreasonable (= 0 or > NumberOfTests). In particular, the user WILL NOT be notified if he uses TSTSET on a Non-(User Test), since this will not affect the result. A negative test number store the inverse value for the test.

Examination of Test Results - Function TSTVAL

IRSLT = TSTVAL (NTEST, IERR)

NTEST = Test number (I*4-input)

IRSLT = Test flag status (L*4-output)

IERR = Error code (2 word I*4 array-output)

Error codes given as IERR (1), IERR (2) below

= 1,2049 success no errors

= -2,2049 illegal test number

Note IRSLT must be declared LOGICAL*4 in the calling program. TSTVAL is typed in the interface statements for the test package.

In order to allow the user to control the program flow based on test results, it is necessary to have a means of examining the results of a given test. While the user could do his own examination of the database directly, it is useful to provide a logical function for this purpose. The user specifies a test number (NTEST) and is returned a LOGICAL result (.TRUE. or .FALSE.) and an error flag. The test number may be positive, in which case IRSLT contains the present value of the specified test result flag. The test number may be preceded by a minus sign, in which case IRSLT contains the logical complement of the test flag value. Finally, the test number may be zero, in which case IRSLT is always .TRUE. This usage is consistent with that for specifying tests to be used in conjunction with histograms.

The error flag would only indicate whether the test number given was greater than the maximum allowable test number specified at initialization time. Note that a test not yet performed is considered to be .FALSE. No check is made on whether or not the specified test has actually been defined.

Zeroing Of Test Counters - Subroutines TSZERB and TSZERA

CALL TSZERB (NBLK, IERR)

CALL TSZERA (IERR)

NBLK = Block number (I*4-input)

IERR = Error code (2 word I*4 array-output)

Error codes given as IERR (1), IERR (2) below

= 1,2049 success no errors

= -2,2049 illegal block number

These subroutines are used to zero the test and block counters either for a specified block (TSZERB) or for all tests and blocks (TSZERA). The latter mode (TSZERA) is likely to be the most common usage. These subroutines can be called both from within the program executing the tests or from programs such as those related to starting a new run. The error conditions would be that the specified block has not been defined (TSZERB only).

Test Package Data Architecture

Test Package Data Base

Note: the following is only valid for Version 2 of PC DAQ and beyond. The test package database is stored as a collection of class object in VB and as arrays of user defined variables in FORTRAN. The properties of the VB classes are defined so that the data is always retrieved or stored in the FORTRAN arrays without any other coding intervention. This insures that the VB values are always in sync with the FORTRAN arrays. There are four class objects. They are cTest, cBlock, cGate, and cBox. For each class object, there is a corresponding class collection object. The property of each class object as two components. The string component has the name of the object and which machine it was created on. The numeric component

contains the test definition, value, and count information. Only the numeric information is stored in the FORTRAN arrays. The code below is the VB definition of the data structures. The FORTRAN has similar type structures defined.

Each of the four classes uses the same type structure to define its string components.

```
Type TestLabels
    ShortName As String     ' Test Name
    MachineName As String   ' Machine the Test was created on
End Type
```

The Test definition is:

```
Type TestInfo
    Deleted As Long         ' Deleted flag. <> 0 means ignore test
    Negation As Long        ' Report NOT of test
    Block As Long           ' Block that the test is in
    Count As Long           ' Count of valid tests
    TestType As Long        ' Test type (ie, AND, OR, etc.)
    Value As Long           ' Value of last test (0 or 1)
    NumParameters As Long   ' Number of parameters in test
    Parameters(1 To NumberOfTestParameters) As Double   ' Parameter values
    Created As Date         ' Creation date and time for object
    CollectionIndex As Long ' Index of Test in collection (test number)
End Type
```

The Block definition is:

```
Type TestBlockInfo
    Deleted As Long         ' Deleted flag. <> 0 means ignore test
    Count As Long           ' Count of valid tests
    FirstTest As Long       ' Index to qTestOrder of first index
    LastTest As Long        ' Index in qTestOrder of last index
    Created As Date         ' Creation date and time for object
    CollectionIndex As Long ' Index of Test in collection (test number)
    VariableType As Long    ' Variable type (from calls to tst...)
End Type
```

The Indirect Gate definition is:

```
Type TestGateInfo
    Deleted As Long         ' Deleted flag. <> 0 means ignore test
    Count As Long           ' Count of valid tests
    Index As Long           ' Index in array of number to test
    HistPointer As Long     ' Histogram pointer
    Low As Double           ' Low value of gate
    High As Double          ' High value of gate
    Created As Date         ' Creation date and time for object
    CollectionIndex As Long ' Index of Test in collection (test number)
End Type
Public Const SIZEOFTESTGATE As Long = 5 * 4 + 3 * 8
```

The Indirect Box definition is:

```
Type TestBoxInfo
    Deleted As Long         ' Deleted flag. <> 0 means ignore test
    Count As Long           ' Count of valid tests
    XIndex As Long          ' X Index in array of number to test
    YIndex As Long          ' Y Index in array of number to test
    HistPointer As Long     ' Histogram pointer
    XLow As Double          ' X Low value of gate
    XHigh As Double         ' X High value of gate
    YLow As Double          ' Y Low value of gate
    YHigh As Double         ' Y High value of gate
    Created As Date         ' Creation date and time for object
```

```
        CollectionIndex As Long ' Index of Test in collection (test number)
    End Type
```

Test Data File

The contents of the test package are automatically dumped to disk as part of the data file at the beginning and end of a run. You can retrieve test scripts and values using standard replay retrieval methods. You can also dump/print/view the test values using the Test Screen.

Example of How to Use the Test Package

The following example illustrates how one might incorporate the test package into an analyzer. This would typically be done in three separate phases. Phase one would normally be executed at the start of each run and would be used to zero out all test and block counters. The second phase would be involved in the analysis of each event. In this phase, one would clear out previous values for the test result flags, set any logical values for user tests, execute various blocks of tests, and examine the latest results of the tests just executed. The final phase would typically be performed after a run was completed. In it, one would likely retrieve the values of the Test and Block counters for the run just completed.

Phase One - Zeroing Test and Block Counters.

```
C
C This section would typically be executed at the start of each
C run. If used within PC DAQ data acquisition or replay
C system, this would likely be placed in auxiliary
C programs QRU1 and XRU1 or the uBegOfRunCall code of an analyzer.
C
      Include `qTestPackage.fi
      Include `qTestPackageImport.fi
      INTEGER*4 IERR(2)       !Status Return
C
      CALL TSZERA (IERR)      !zero out all test and block counters
      IF (IERR(1) .NE. 1) then      !List if error occurs
            Error response code
      end if
C
C Here you would do other things you might do at the start of a run
C such as zeroing out scalars.
C
C THIS COMPLETES THE FIRST PHASE.
C
```

Phase Two - Test Execution

This code would normally be executed once for each event.

```
      Include 'qTestPackage.fi'
      Include 'qTestPackageImport.fi'
      INTEGER*4 IDW(100)      !Data array to be tested
      INTEGER*4 IERR(2)  !Status return
C
      CALL TSCLFA(IERR)  !Zero out all test result flags
      IF (IERR(1) .NE. 1) then !List if error occurs
            Error response code
      end if
C
```

```
      CALL TSTSET(10,.TRUE.,IERR) !Preset Test 10 result = .TRUE.
C
C Note that the value of test 10 does not actually become .TRUE.
C until test 10 (which must also be a USER Test) is
C executed as a result of a call to TSTEXE.
C
      IF (IERR(1) .NE. 1) then !List if error occurs
            Error response code
      end if
C
      CALL TSTEXE(1,IDW,IERR)  !Execute Block 1 of tests
      IF (IERR(1) .NE. 1) then !List if error occurs
            Error response code
      end if
C
      IF (TSTVAL(-10,IERR))GO TO 100 !Skip to 100 if test 10=.FALSE.
C
C This would be code you want to skip if test 10 is .FALSE.
C
100   CONTINUE
      IF (IERR(1) .NE. 1) then !List if error occurs
            Error response code
      end if
C
C THIS ENDS SECOND PHASE.
C
```

Phase Three - Retrieving Counters

This code would normally be executed after a run is complete. It could perhaps be performed within the analyzer in response to uEndOfRunCall.

```
      Include `qTestPackage.fi
      Include `qTestPackageImport.fi
      INTEGER*4 IERR(2) !Status return
      INTEGER*4 TCNT(NumberOfTests) !Array to hold test counters
      INTEGER*4 BCNT(NumberOfTestBlocks) !Array to hold block counters
      INTEGER*4 TPARAM(24) !Array for Test Parameters
C
      IF (IERR(1).NE.1) GO TO 100
      CALL TGTPRM(TPARAM,IERR) !Get test data base parameters
      IF (IERR(1) .NE. 1) then      !List if error occurs
            Error response code
      end if
      IF (IERR(1).NE.1) GO TO 100
C
      CALL TCTCOR(1,NumberOfTests,TCNT,IERR) !Get test counters
      CALL BCTCOR(1,NumberOfTestBlocks,BCNT,IERR) !Get Block counters
C
C You may now do whatever calculations you want using the values of
C the Test and Block counters.
C
C Note that if you simply want a list of the values this is a waste
C of effort, since program TPR provides that function.
C
C
100   CONTINUE
```

```
C
C THIS ENDS PHASE 3.
```

Setting up Indirect Gates and Boxes

It is possible to set the values for the data word indices and limits for indirect gates and boxes from within plot display program. The determination of whether you are referring to a gate or a box is taken from whether a one-parameter or a two-parameter histogram is being displayed.

Indirect Gates and Boxes are interactively defined using the Zoom function for plots. The steps involved are:

- Display a graph/plot that shows the variable(s) on which you want to define a cut.
- Be sure the graph has the focus.
- Use the Plot / Pick Zoom/Gate… menu option to pick which plot on the graph to use.
- Use the Plot / Gate/Box… menu option to define a gate. The program will ask for a gate (box) number. Give it the appropriate number. Then use the mouse zoom to define a region on the plot. Click one extreme of your choice and, holding the left button down, drag the definition box to the other extreme. Then lift up on the mouse button. That is it. You can view or alter the results by looking at the gate (box) on the edit form. Note, the program will check that the variable you have plotted is in the same system block as other tests in the block that has a test that refers to you new gate.

System Requirements

Computer

High speed Pentium or Pentium Pro. The faster, the better. It will work on 486's, but only at a walk.

64 megabytes of RAM minimum.

>1 GB disk

CD-ROM.

High-end graphics card (support for 64K colors at high resolution).

Network card. I am putting 3COM 100base-TX cards on my computers.

GPIB or other DAQ hardware interface.

System

Windows 95®, Windows NT 4.0® with SP3 or later, or above. I recommend NT, but you need to check what hardware is on your computer. Windows NT is somewhat more restrictive about what it can use. Plug 'n Play is only supported by Windows 95 at this time. Trials with Win 95/98 have been very limited. This program has been used in DAQ mode only on NT machines. Replay has been demonstrated on both Windows 95® and Windows NT®

Software

You must have FORTRAN to write CAMAC and analysis code. Visual Basic is needed only if you want to be able to change control code. C++ is not currently used. I will shortly (as of 10/98) be upgrading to the Visual Studio 6 version of the languages.

DEC Visual Fortran 5.0. Note, code from the Numerical Recipes books in included.

MS Visual Basic 5.0. Enterprise edition

MS Visual C++ 5.0. Enterprise edition

MS Office 97. EXCEL is useful to analyze plots and is used by the variable management tool.

If you are buying new licenses for many of the above items, it may be cheaper to buy Microsoft Developer Network Universal Subscription. This has the system, C++, VB, and Office with lots more. Quarterly updates. No paper manuals, it is all on the CDs. $2500.

CAMAC Interface

The following can read CAMAC at >400 KB/s in DMA mode (ex., FERA Memory module). Single CAMAC reads are much slower. For single reads, 1 kHz on a 166 MHz Pentium is possible. Block reads of ADCs and TDCs can speed things up. DMA speeds are nearly independent of CPU speeds. Single reads scale as CPU speed.

National Instrument AT-GPIB/TNT card and driver (part 776836-01) with NT software. Also available with Windows 95 drivers. Cost: 500

Kinetics Systems GPIB Crate Controller. 3988-G3A. Cost: 2696.42. Note, this can be a long lead time item.

LAMPF Event Trigger Module.

Tape

I use a 4mm DAT drive on a SCSI controller to write to tape. Taping is done as a periodic backup of the disk. That is because Windows only handles tapes as backup devices.

Tape drive. 4mm DAT seems to work fine. If you need to push the 400Kb/s limit, a fast tape/disk technology should be considered.

SCSI adapter if not already represent. Note, all adapters are not compatible. Compare tape and adapter specs. If only used for taping, a low end SCSI-2 adapter is good enough.

Backup software. I use Seagate Backup Exec (formally Arcada). You might get by with the backup software that comes with NT.

Format Standards for PC DAQ

Format Standards

This section describes various format standards for the PC DAQ system. The first section describes "scripts." A script is an ASCII command file providing control information. The next section describes the database. The last section describes the output format.

Blocks are data arrays. Up to 40 blocks can be defined for histogramming purposes. Only three are predefined. They are the "System Blocks." The first block is a large 16-bit integer array [IntegerData ()]. The second is a 32-bit single precision real array [SingleData ()]. The third is a 32-bit integer array [LongData ()]. Definition and use of other blocks is the user's responsibility. There is a NULL data value defined for each system block. At the beginning of every run (or batch of runs if in auto restart or replay batch mode), the system blocks are cleared to the NULL value before any user routines are called. NULL data is not displayed or histogrammed. It is written to "tape" if present.

Scripts

Scripts are ASCII control files. They currently are limited to 64K bytes apiece. There are many kinds of scripts. They include:

1. Retrieving Scripts
2. Run Control [RunControl]
3. Labels [Labels]
4. Histograms [Histograms] and [Ahistograms]

   Time Series: name /switches

   1-D and 2_D frequency plots: name /switches
5. Tests [Test]
6. Plots [Plot]
7. Graphs [Graph]
8. Comments [Comment]
9. End of File [EOF]

Scripts have certain things in common

Identifier line. This is a string enclosed in brackets such as [RunControl]. Only the first three letters are used to identify a script.

Variable reset. Starting a command section usually resets all variables defined by that section to some default value. For example, a new histogram script [HIST] automatically erases all previously defined histograms. The [AHI] command does not reset the histogram list, but rather appends histograms to the existing list.

Comment lines begin with a ";" in column 1. Inline comments start with a "!".

Case insensitive command lines.

Names, such as Histograms names, preserve case. If you need to put a special character (;!/:) in a name, enclose the name in double quotes.

Special characters are available for histogram names. Upper and lower case and Greek characters may be drawn (see the Tedi manual or appendix I of the plasma physics plotting package manual.) as in Tedi. A "second" font character is specified by preceding its "first" font equivalent with an \.

The ^ character moves everything that follows up 1/2 space while the _ character lowers everything that follows by 1/2 a space. The $ character causes a backspace. The @ character specifies that one of these special characters is to be the actual character rather than a flag. Thus to obtain an alpha use the string '\a' and to obtain an \ use the string '@\'. To obtain a superscript 2 use either '\2' (small) or'^2_' (full size). Note: one must use '@@' to obtain an actual @ character.

Switches force special actions to be taken. The general format of a switch is:

/SWitch:val_1:val_2:...:val_n

The delimiting slash and the two (sometimes one) characters must be given if the switch is given at all. The following characters may be appended for clarity but are not required. For example, histogram scripts has a switch "/Event" which specifies which event goes with a histograms.

/Ev:13

/Eve:13

/Event:13

all work

The quantities "val_j" are input values. They may be numbers or alphanumeric strings. The values must be separated by colons as shown. The number and type of values required are described with each switch. Except where noted, numeric values are decimal integers.

Some switches have defined inverses. These are denoted by placing a minus sign between the slash and the first character. For example, if "/LO" means "write output," "/-LO" means "do not write output."

Optional Values. In the descriptions that follow, items in square brackets are optional.

Scripts can all be in one file, or in separate files. Using separate files for each script type is recommended. For each script type, there is an edit control form. Each form has “Generate Script” button. Clicking this button will generate an example script describing the current setup into the "Replay Script" text box.

Automatic scripts recall. When you start PC DAQ, it will automatically reload any control (non-plot) scripts that were open when the program was last terminated. It will then ask you if you want to run them immediately. Finally, it will ask you if you want the run control started. This will reduce the number of mouse clicks to start the program from about ten to two. It also makes the "separate files" method of script management mentioned above easier to handle. Also, when windows are minimized, an easier to read caption will be displayed.

Retrieving Scripts

At the beginning of every output file is a section of data that contains all the script information used to control that run. This includes the run controls, labels, histograms, and test definitions. To see what those values were, the easiest procedure is to do:

Execute normal scripts. Read in and execute your normal scripts for the file in questions.

Change Replay Options. Under the menu “Options,” chose Replay… . This will bring up the Replay options control. Click off the box Data. This will turn off data reads. Then click on Labels, Histogram Definitions, Controls, and Test Definitions.

Run the file. Pick your replay file and run it. After you see the current file position moving, hit the stop.

Generate scripts. The program will have already dumped the run controls to the Replay Script form. To generate the other scripts, go to the menu “Options / Labels…” . This will bring up the label definition form. At the bottom of the form is a Generate Script button. Click it, and a [Label] section will be appended to the Replay Script form. Do the same for Histograms and

Tests. You have now regenerated the original script! If you want to reuse this, cut and paste this to another script form and modify as desired.

Run Control [RunControl]

Run control is used to set the enable flags for run control and events. It uses a Windows INI like format (descriptor =[value [, value [, …]]]). For Yes/no values, only the first letter is examined. The following are equivalent values for "Yes" = {y | Y | t | T | 1 | x | X}. Anything else is a NO. Most of these commands are illegal when a run is in progress. Some require that you "Disconnect" before using. Items in square brackets are optional.

This information is saved at the beginning of each run in the control section if output is enabled. If the input is from DAQ, the replay control values are not dumped. If the input is a disk file, the CAMAC control functions are not dumped. Included in this data are the machine name, the user's account, and the PC DAQ VB code version numbers, which are not definable by the user.

See Retrieving Scripts for details on how to view script information written to disk.

Defined commands are:

AllowRemote*[= flag]* :: Allow remote users. Unspecified default is no.

ClearTestsBeginOfBatch *[= flag]* :: Clear all test blocks at the begin of a batch of runs. Unspecified value is no.

ClearTestsBeginOfRun *[= flag]* :: Clear all test blocks at the begin of a run. Unspecified value is no.

CAMACTriggerType = *nn* :: Type of CAMAC module that is generating trigger. 1 is LAMPF trigger module and is the default.

CrateEnable = *crate,GPIB address* :: Enable crate at given GPIB address. Valid crate numbers are 1 through 7.

DatabaseEnable *[= flag]*:: If flag is yes, use database files for calibration and definitions. Unspecified value is yes.

DatabasePath *[= path]* :: Path for database files. The default path is "..\Databasefiles".

DatabaseFile = *nn,filename* :: Define file names for database files. Up to seven files maybe defined.

EventModule = *crate, slot* :: Crate and slot of (LAMPF) Event Trigger Module.

HistEnable = *block[,autofill]* :: Block = enable histograms for block number "block". The first three blocks are the system blocks. For the system blocks only, autofill = yes tells the control program to do the incrementing if the data does not equal the NULL data value.

HistSave = *[-]block* ::. Save histograms for block "block" at end of run. Leading minus means do not save.

IssueCZ *[= flag]*:: If flag is yes, issue a C and Z to all CAMAC crates at initialization time. Unspecified value is no.

LogEnable *[= flag]* :: If flag is yes, write data to tape. Applies to both DAQ and Replay runs. Unspecified value is no.

OutputPath*[= path]* :: Path for output files. The default path is "..\Outputfiles".

ReadQX *[= flag]*:: If flag is yes, return Q and X for every FCNA command. Unspecified value is no.

ReplayControl = *flag* :: If Flag = yes, then the control settings in the replay file are written to the replay script file. Otherwise they are ignored, as by definition, a run is in progress. To use the control setting, you need to save the replay script to a file, then start a new run with this as the command script. Be sure you set the ReplayFlag command to yes and the ReplayType to P in the new script.

ReplayFile = *filename* :: Name for replay file. If no path given, use ReplayPath.

ReplayFlag*[= flag]* :: If flag is yes, the run is replay; if MC, then it is Monte Carlo; if SM, then it is Sample Mode; else the run is DAQ.

ReplayHistBeginDef *[= flag]* :: If Flag = yes, then histogram definition information is taken from the beginning of the replay file. Any histogram definition information that had been read in from a script is ignored. All histogram data is lost. Unspecified value is no.

ReplayHistEndDef *[= flag]* :: If Flag = yes, then histogram definition information is taken from the end of the replay file. Any histogram definition information that had been read in from a script is ignored. All histogram data is lost. Unspecified value is no.

ReplayHistData *[= flag]*:: If Flag = yes, then histogram data information is taken from the replay file. Any previous histogram data or definitions are lost. This flag is automatically set if the file extension is ".hsv". Unspecified value is no.

ReplayLabels *[= flag]* :: If Flag = yes, then label definition information is taken from the replay file. Any label information that had been read in from a script is ignored. Unspecified value is no.

ReplayPath*[= path]* :: Path for replay files. The default path is "..\Outputfiles".

ReplayResetHistDef *[= flag]* :: If Flag = yes, then the histogram definition list is reset before reading in histogram information from the replay file. Set this = n if you want to merge your script histogram list with histograms in an HSV or replay file. Unspecified value is no.

ReplayTestDef*[= flag]* :: If Flag = yes, then test definition and count information at the beginning of a run is taken from the replay file. Any test information that had been read in from a script is ignored. Unspecified value is no.

ReplayTestDump *[= flag]* :: If Flag = yes, then test definition and count information at the end of run is taken from the replay file. Any test information that had been read in from a script is ignored. Unspecified value is no.

ReplayType*[= type]* :: Type of replay file. P or none means PC DAQ format. E means MEGA EMS. Q means VMS Q.

Restart*[= interval]* :: If interval present, then enable automatic restart of DAQ runs after a fixed time interval (units are minutes). If interval not present, disable automatic restart.

Title = *Comment* :: Run title.

GENERAL PERMITS FOR EVENTS

BeginEvent = ***nn[,Title]*** :: Begin a new event with event number nn. This is required for every event. Must be the first line for each new event. Note, if you use a number higher than 24, you will need to extend the Select statement in Analysis.for and create new Analxx.for routines. The maximum number is set by the NumberOfEvents parameter in PCDAQBuffers.fi. Talk to the author if you need to go above 99. Event 0 is defines the global permit for a given action. Title gives a name to the event. The default title is "Event nn" for a specific event, "Global" for the global event. A user supplied global event title is ignored.

EndEvent :: Finished definition of a single event.

AsynchRead = ***flag,[ErrSuspend, [TimeOut, [PollTime]]*** :: Enable asynchronous reads if flag is yes. TimeOut is the time to wait before declaring failure. PollTime is the time between checks of the completion flag. Times are in seconds. The default is AsyncRead = no,no,1,0.1. The run loop continues until the complete flag is turned on or the read times out. If it does not complete and ErrSupend is yes, then the run will be suspended. If ErrSupsend = no, the program will continue without an error message and will continue to poll until the end of the run, but with a polling interval of TimeOut instead of PollTime. When it does complete, the analysis routines will be called. This parameter is not meaningful for the global event.

ClearHistBatch *[= flag]* :: If flag is yes, then at the beginning of every batch of runs, all histograms associated with this event are cleared. The unspecified default is yes for both global and specific events. Both the global and the specific event must be true for a clear to be done. If all defined events are set to clear, then all histograms are cleared, even if they are associated with undefined events.

Defined means that a BeginEvent exists for the event in the script. Automatic histogramming must also be enabled.

ClearHistRun ***[= flag]*** :: If flag is yes, then at the beginning of every run, all histograms associated with this event are cleared. The unspecified default is no for specific events, yes for the global event. Both the global and the specific event must be true for a clear to be done. If all defined events are set to clear, then all histograms are cleared, even if they are associated with undefined events. Defined means that a BeginEvent exists for the event in the script. Automatic histogramming must also be enabled.

Display = ***flag,TestNumber[,Suspend]*** :: Automatic update of single event displays. Flag enables automatic display updates. Test number is the number of the test that must be true for the display to update. Suspend tells whether the run should be suspended when the update is done. Default value is Display=n,x,n. Not meaningful for the global event.

DumpControl ***[= flag]*** :: If Flag = yes, then the control settings in the replay file are written to the replay script file for this event. The DumpControl flag for the global event and ReplayControl must also be set before this will have an effect. Default value is yes.

EndOfRunTrigger ***[= flag]*** :: If flag is yes, then an event trigger is generated by a normal end of run command (either rollover or manual). Default is no for both global and specific events.

MayProcess = ***interval*** :: If must process is not set, every (interval)th trigger is analyzed. Default interval is 1. Only directly effects events that are hardware triggered. Once a hardware trigger is sent for analysis, all software-triggered events associated with it are automatically called. For analysis programs that use the recursion feature, may process means that a recursion request will be ignored if new data is available if in DAQ mode. In replay mode, once analysis is begun, it continues until all recursion requests are handled. Not meaningful for the global event.

MustProcess ***[= flag]*** :: If flag is set, event is must process. Every event is analyzed. For analysis programs that use the recursion feature, must process means that a recursion request will be honored even if new data is available. Default is yes. Not meaningful for the global event.

RefreshLimit = ***nn*** :: The number of times the input is checked for data between scans for user control changes (i.e., mouse clicks). Default is 10 for both global and specific events. The lesser of the global and specific event values are used.

SaveHistBatch ***[= flag]*** :: If flag is yes, then at the end of every batch of runs, all histograms associated with this event are saved. Default is yes for both global and specific events. Both global and specific events must be true for save to be done. If all defined events are set to save, then all histograms are saved, even if they are associated with undefined events.

SaveHistRun ***[= flag]*** :: If flag is yes, then at the end of every run, all histograms associated with this event are saved. Default is yes for both global and specific events. Both global and specific events must be true for save to be done. If all defined events are set to save, then all histograms are saved, even if they are associated with undefined events.

Source = ***source*** :: Source = trigger source. 0 = means hardware (CAMAC or Replay input data), otherwise it is a software trigger initiated by the “source” event number. Default is zero for both global and specific events. Not meaningful for the global event.

Suspend = ***flag,TestNumber*** :: If flag is yes, then the run will automatically suspend when the test becomes true. Default is flag=false. Not meaningful for the global event.

TimeTrigger = ***flag[,Interval]*** :: Enable a trigger that runs at a fixed interval. Interval is in seconds. Default is TimeTrigger=n,60. The time is not meaningful for the global event.

UpdatePlotBatch ***[= flag]*** :: If flag is yes, then at the end of every batch of runs, all plots associated with this event are updated on the screen. Default is yes for both global and specific events. Both global and specific events must be true for update to be done. If all defined events are set to update, then all histogram plots are updated, even if they are associated with undefined events. Event association is through the histograms in the plot.

UpdatePlotRun ***[= flag]***:: If flag is yes, then at the end of every run, all plots associated with this event are updated on the screen. Default is yes for both global and specific events. Both global and

specific events must be true for update to be done. If all defined events are set to update, then all histogram plots are updated, even if they are associated with undefined events. Event association is through the histograms in the plot.

DEFAULTS:

Table 1, General Event Defaults

Name	Global Default	Specific Default
Asynchronous Error Suspend		n
Asynchronous Polling Time (s)		0.1
Asynchronous Read Enable		n
Asynchronous Time Out (s)		1
ClearHistsBatch	y	y
ClearHistsRun	y	n
Display Enable		n
Display Suspend		n
Display Test Number		0
Dump Control	n	y
May Interval		1
Must Process		y
Refresh Limit	10	10
SaveHistsBatch	y	y
SaveHistsRun	y	y
Title	Global	Event n
Suspend Test Enable		n
Suspend Test Number		0
UpdatePlotsBatch	y	y
UpdatePlotsRun	y	y

DETAILED PERMITS FOR EVENTS

EventEnable ***[= flag[,detail[,detail[,…]]]]*** :: If flag is yes, the event is enabled. Detail breakdown by function can be defined. See later explanation of details.

UserEnable ***[= flag[,detail[,detail[,…]]]]*** :: If flag is yes, calls to the user analysis program ANALnn during data taking are enabled. See later explanation of details. The following parameters control the calling of the ANALnn as follows:

Table 2, Permits for Calls to Analnn

Permit	Parameter	Mnemonic	Value	Comment

UserEnable	DN	uNewCall	0	Data has changed to new values
UserEnable	DL	uRecursiveCall	1	Analysis is being called again
UserEnable	BR	uBegOfRunCall	2	Analysis is being called at start of run
UserEnable	ER	uEndOfRunCall	3	Analysis is being called at end of run
UserEnable	DC	uClearCall	4	Analysis is being called at end of analysis loop
CamacEnable	DR	uGetTrigger (Fortran)	5	Analysis is being called to get raw Data
VBEnable	HT	uGetTrigger (VB)	5	Scan Active X servers for trigger
MCEnable	DR	uGetMonteCarlo (Fortran only)	6	Generate Monte Carlo Data
UserEnable	BS	uBegOfSessionCall	7	Analysis is being called at start of session
UserEnable	ES	uEndOfSessionCall	8	Analysis is being called at end of session
UserEnable	BB	uBegOfBatchCall	9	Analysis is being called at start of batch
UserEnable	EB	uEndOfBatchCall	10	Analysis is being called at end of batch
UserWrite	BS	uBegOfSessionWrite	11	User write is being called at start of session
UserWrite	ES	uEndOfSessionWrite	12	User write is being called at end of session
UserWrite	BB	uBegOfBatchWrite	13	User write is being called at start of batch
UserWrite	EB	uEndOfBatchWrite	14	User write is being called at end of batch
UserWrite	BR	uBegOfRunWrite	15	User write is being called at start of run
UserWrite	ER	uEndOfRunWrite	16	User write is being called at end of run
UserWrite	DW	uDataWrite (Fortran)	17	User write is being called at end of each event analysis
VBEnable	DW	uDataWrite (VB)	17	Write data to Active X Servers in replay mode at the end of the analysis
UserRead	BS	uBegOfSessionRead	18	User Read is being called at start of session
UserRead	ES	uEndOfSessionRead	19	User Read is being called at end of session
UserRead	BB	uBegOfBatchRead	20	User Read is being called at start of batch
UserRead	EB	uEndOfBatchRead	21	User Read is being called at end of batch
UserRead	BR	uBegOfRunRead	22	User Read is being called at start of run
UserRead	ER	uEndOfRunRead	23	User Read is being called at end of run
UserRead	DR	uDataRead (Fortran)	24	User Read is being called at start of analysis
VBEnable	DR	uDataRead (VB)	24	Get data from Active X Servers. DAQ or Monte Carlo modes only?

SimpleEnable ***[= flag[,detail[,detail[,…]]]]*** :: If flag is yes, calls to the simple analysis transformation programs for the event are enabled. See later explanation of details. Only the DN parameter is used. It applies to both new and recursive analysis calls.

AutoWrite ***[= flag[,detail[,detail[,…]]]]*** :: If flag is yes, automatic output for the event is enabled. See later explanation of details.

AutoRead ***[= flag[,detail[,detail[,…]]]]*** :: If flag is yes, automatic input for the event is enabled. See later explanation of details.

UserWrite ***[= flag[,detail[,detail[,…]]]]*** :: If flag is yes, user output for the event is enabled. See later explanation of details.

UserRead ***[= flag[,detail[,detail[,…]]]]*** :: If flag is yes, user input for the event is enabled. See later explanation of details.

AutoHist ***[= flag[,detail[,detail[,…]]]]*** :: If flag is yes, automatic histogramming for the event is enabled. See later explanation of details.

UserHist ***[= flag[,detail[,detail[,…]]]]*** :: If flag is yes, user histogramming for the event is enabled. See later explanation of details.

CAMACEnable ***[= flag[,detail[,detail[,…]]]]*** :: If flag is yes, CAMAC hardware triggers for the event are enabled. See later explanation of details. HT enables the hardware interrupt device, DR enables the call to ANALnn(uGetTrigger).

ManualEnable ***[= flag[,detail[,detail[,…]]]]*** :: If flag is yes, manual triggers for the event are enabled. Manual trigger buttons are at the bottom of the main control form. See later explanation of details. Only HT (hardware trigger) has meaning.

VBEnable ***[= flag[,detail[,detail[,…]]]]*** :: If flag is yes, Visual Basic data and/or triggers for the event are enabled. See later explanation of details. These VB calls are different from and in addition to the calls to ANALnn. Typical uses for the parameters are:

BS :: Connect to ActiveX servers. Turn off the servers' access to GPIB if on the buss as the CAMAC crate.

ES :: Disconnect from ActiveX server. Turn their access to GPIB back on.

DR :: Get data from an ActiveX server.

DN, DL :: Call an analysis code in written VB.

DW :: Send data to an ActiveX server. Usually called only in replay mode.

DC :: Whatever hits you as useful.

HT :: See if an ActiveX server has raised a trigger flag.

DT :: The same as HT?

MCEnable ***[= flag[,detail[,detail[,…]]]]*** :: If flag is yes, Monte Carlo triggers for the event are enabled. See later explanation of details. HT is generated with each pass through the run control loop. DR calls ANALnn (uGetMonteCarlo) to generate an event.

OtherEnable ***[= flag[,detail[,detail[,…]]]]*** :: If flag is yes, triggers from another source (to be defined) for the event are enabled. See later explanation of details. This is supplied to provide for future expansion either by the user or the great and wise PC DAQ design team.

ReplayEnable ***[= flag[,detail[,detail[,…]]]]*** :: If flag is yes, replay triggers for the event are enabled. See later explanation of details. If Flag = yes, then the event data is read into the appropriate buffers for enabled events and have a current trigger source of 0 (zero). A "hardware" trigger for the event is generated for the analysis routines are called once all data for a specific event is read in. Data goes back into the same locations from which it was written. If may process is selected and the interval is not 1, analysis will be skipped as appropriate. Once analysis is begun, it will continue until all new recursion requests are handled. If different events have been written out for the same recursion pass on tape, each event will generate a separate set of calls to the analysis routines. If the same event occurs in multiple recursion passes as recorded on tape, then its analysis routine will be called for each recursion pass in which it occurs. If the current setting for the trigger source is not zero or the event is not enabled, then the data on tape is ignored (skipped). If the ReplayEnable flag = no, then all data from that event is ignored. Do this to quickly get to the end of run data or histogram data.

Parameter values mean:

BB, BR :: If the parameter is turned on, then the event begin of batch/run data is read into the appropriate buffers for enabled events when in replay mode. Note that as this comes after the actual calls to the begin of batch/run analysis routines, this replay data will overwrite any information those routines have placed in the system blocks. Data goes back into the same locations from which it was written.

EB, ER :: If the parameter is turned on, then the event end of batch/run data is read into the appropriate buffers for enabled events. Note that as this comes before the actual calls to the end of run analysis routines, this replay data will be overwritten by any information from those routines placed in the system blocks. Data goes back into the same locations from which it was written.

DR :: Enables reading in of regular data buffer into the data buffers.

HT :: Allows the reading in of data to raise a trigger. Does no raise a trigger for a sub-trigger event unless DT is also present.

DT :: Overrides the “Source” sub trigger parameter. Instead of being triggered as a sub-trigger, the event is triggered by replay data being read in.

Permit Detail flags. Each permit can be subdivided into many pieces, each of which controls a different part of the data taking cycle. Not all parameters are meaningful for all permits. See the table below for what parameters are defined for which permits and their default values. See the definition of each permit for clarification of what a parameter means to a given permit. A leading minus sign disables the step.

- ***[-]BS*** :: Begin Session. Actions taken just before hardware connections are made. For instance, connections to other ActiveX servers by VB programs go here.
- ***[-]BB*** :: Before Batch. The beginning of a group of runs. For live data, the start of data taking when rollover is enabled.
- ***[-]BR*** :: Begin Run. The start of each run.
- ***[-]ES*** :: End Session. Actions taken just after hardware connections are broken. For instance, disconnections to other ActiveX servers by VB programs go here.
- ***[-]EB*** :: End of Batch. The end of batch of runs.
- ***[-]ER*** :: End of Run. The end of each run.
- ***[-]DR*** :: Data Read.
- ***[-]DN*** :: Data New. Start of new recursive loop.
- ***[-]DL*** :: Data Loop. A recursive data analysis call.
- ***[-]DW*** :: Data Write.
- ***[-]DC*** :: Data Cleanup. Cleanup after an analysis call.
- ***[-]HT*** :: Hardware Trigger.
- ***[-]DT*** :: Data Trigger.

The following table shows which parameters apply to individual permit types. A gray box means the parameter has no meaning. The letter “y” or “n” is the default value of the parameter if the permit is not in the script. If you put the permit name in the script with no values attached, the master (flag) value is assumed to be true.

Table 3, Permit Defaults

Default Event	Event Enable	User Enable	Simple Enable	Auto-Write	Auto-Read	User-Write	User-Read	Auto Histogramming	User Histogramming	Command Enable	Manual Enable	Write Enable	Monitor Gate Enable	Other Enable	Replay Data Enable
Master (flag)	n	n	n	y	y	n	n	y	n	n	y	n	n	n	
Begin Session	n	n										y			
Begin Batch	y	y		y	y	n	n	y	n			y			
Begin Run	y	y		y	y	n	n	y	n			y			
End Run	y	y		y	y	n	n	y	n			y			
End Batch	y	y		y	y	n	n	y	n			y			
End Session	n	n										y			
Data Read	y				y		n			y		y	y	y	
Data Analysis New	y	y	y									n			
Data Analysis Recursive	y	y										n			
Data Write	y			y		n		y	n			y			
Data Clear	n	n										n			
HardwareTrigger	y									y	y	n	y	y	
DataTrigger	n											n		y	
End of Run Trigger	n														
TimeTrigger	n														

Labels [Labels]

Labels are used to define the contents of the system blocks. They assign names to locations in the system blocks, define display formats for the data screen, define simple data transformations, and control the I/O for the events at the individual variable level. The run must be stopped or suspended for label changes to be made. Labels with names of the form CAMACnn are used to defined the location of CAMAC I/O buffers for event nn within the system blocks. Unlike other label defined arrays, CAMAC I/O buffers are considered to be of variable length w.r.t. I/O operations. Internally, current locations and sizes of CAMAC buffers are kept separately from other label information. To perform automatic taping from a defined label location for an event, you must enable both the specific event output flag and the output flag for the variable. A given label is associated with both a specific event and a specific system block. The best way to generate your Label command section is with the variable management tool.

See Retrieving Scripts for details on how to view script information written to disk.

Label format: name /switches

/[-]DF :: Use command line to define/undefine defaults. Defines label $default$.

/[-]LO :: Write variable as part of event output buffer. Event output and automatic event output flags must also be set. Note that for CAMAC buffers, logging to tape only occurs once for a given trigger. For multiple calls of the event (due to recursion requests), only non-CAMAC variables are taped for the second and following calls.

/BL:*bx* :: Block index for label. Only system blocks are valid.

/IN:*index[:size]* :: Index to system block. Size of array (default = 1). If a CAMAC buffer, size is maximum allowed size of buffer.

/FO:*{I|H|O|B|T|E|n}* :: Display format. I = integer, H = hex, O = Octal, B = Binary, T = True/False, E = Exponential, n = fixed decimal [n digits to the right of decimal]

/EV*:event* :: Event number associated with label.

/**FM*:n*** :: Simple transform to use to generate value for label. 0 = none, 1 = linear transform [value = offset + slope * input], 2 = first user defined transform (SimpleUser1), 3 = second user defined transform (SimpleUser2)

/SL*:slope* :: Slope for linear transform. First parameter for user defined transform.

/**OF*:offset*** :: Offset for linear transform. Second parameter for user defined transform.

/IB*:block* :: Block number of input to simple transforms. Must be a system block.

/II*:index* :: Index of input to simple transform.

/NA*:Comment* :: Comment

Histograms [Histograms] and [Ahistograms]

Histogram definitions can be done in two types of sections. When a [HIS] section is encountered, the program resets the defined histogram list and starts afresh. When a [AHI] section is found, histograms are appended to the current list of defined histograms.

Time Series: name /switches

/[-]DF :: Use command line to define/undefine defaults. Defines histogram $default$.

/[-]LO :: Write data to tape at end of run. Block save flag must also be set.

/TI :: Histogram is a time series histogram.

/BL:*bx* :: Block index for histogram input. Only system blocks are valid.

/IN:*ix* :: Index to block array.

/EV:*event* :: Event number associated with histogram.

/TE:*n*:*use* :: n is the test number to use. If n = 0, the test is ignored. If n is positive, then test n of the test package must be true for the histogram to be incremented. If n is negative, then test n must be false. If use = 0, then the test is ignored.

/X:*min*:*max* :: Default minimum and maximum histogram values for plot.

/XD:*min*:*max*:*show* :: Limit lines. Min and Max values. Show = 1 means show on plot. The limit (warning) lines that can be defined in the histogram definition are now drawn on the plot if the grid control is turned on. Use the "User 1" grid control for the lower limit, the "User 2" control for the upper limit. See the Grid tab on the Edit Plot form.

/XL:*label* :: X axis label.

/XP:*ix*:*min*:*max* :: Index to block array. Default minimum and maximum histogram values for plot.

/NA:*Legend* :: Legend text to use for overlays.

1-D and 2_D frequency plots: name /switches

/[-]DF :: Use command line to define/undefine defaults. Defines histogram $default$.

/[-]LO :: Write data to tape at end of run. Block save flag must also be set.

/-TI :: Histogram is a frequency histogram.

/BL:*bx* :: Block index for histogram input.

/IN:*ix[:iy]* :: Index to block array.

/EV:*event* :: Event number associated with histogram.

/TE:*n*:*use* :: Tests. N is test number to use. If use = 0, then the test is not applied.

/MU:*nn* :: Histogram is a multiple entry histogram if nn > 0. First value at the array index is the number of entries that follow. nn is the maximum number of entries. Histograms then use nn+1 words in the block.

/BI:*nx[:ny]* :: Bin size.

/X:*min*:*max* :: Minimum and maximum histogram values.

/XD:*min*:*max*:*show* :: Limit lines. Min and Max values. Show = 1 means show on plot. The limit (warning) lines that can be defined in the histogram definition are now drawn on the plot if the grid control is turned on. Use the "User 1" grid control for the lower limit, the "User 2" control for the upper limit. See the Grid tab on the Edit Plot form.

/XL:*label* :: X axis label.

/XP:*ix*:*min*:*max*:*nx* :: Index to block array. Minimum and maximum histogram values. Bin Size.

/Y:*min*:*max* :: Minimum and maximum histogram values.

/YD:*min*:*max*:*show* :: Limit lines. Min and Max values. Show = 1 means show on plot. The limit (warning) lines that can be defined in the histogram definition are now drawn on the plot if the grid control is turned on. Use the "User 1" grid control for the lower limit, the "User 2" control for the upper limit. See the Grid tab on the Edit Plot form.

/YL:*label* :: Y axis label.

/YP:*ix*:*min*:*max* :: Index to block array. Minimum and maximum histogram values. Bin Size.

/NA:*Legend* :: Legend text to use for overlays.

Tests [Test]

Test package. Based on the Q test package document MP-1-3412-3, Rev: March 17, 1986.

/te:*n*, /bl:*n*, /ig:*n*, /ib:*n*. :: Ignored. Formally used to set maximum number of tests, blocks, gates, and boxes. Maximums are set as parameters in both VB and FORTRAN codes.

bl,*n* :: This line is used to define the beginning of the test descriptors for a given block of tests. All test descriptor lines from this point until the next block delimiter or end of file will be taken as part of the specified block. N is a block number. It should be noted that this line is NOT in the /SWitch format.

nn,*type,n1,n2,n3* :: Test definition for test nn. Test type is “type”, allowed values are: Bit, Equal, Pattern, Gate, Igate, Ibox, AND, IOR, EOR, Majority, and User. N(i) are parameters controlling the test.

/Gate:*ngate[:index:low:high]* :: Indirect gate limit definition.

/Box:*nbox[:xindex:xlow:xhigh:yindex:ylow:yhigh]* :: Indirect box limit definition.

Plots [Plot]

For internal use only. These commands record how a histogram is displayed.

Graphs [Graph]

For internal use only. These commands record how a page of plots is displayed.

Comments [Comment]

Anything can go into here except a section heading (i.e., a line starting with a square left bracket [ ).

End of File [EOF]

Optional end of file marker. Any data after this is ignored.

Database

The database is a set of ASCII flat files containing constants, geometry definitions, and analysis calibration values keyed run number. This includes data maps, pedestals, gains, and various other constants needed to decode and analyzing an event. This database program is adapted from the MEGA database (LAMPF experiment 969).

Database General

Event 1 is used to read in the database values. At the start of each run, all enabled events are called with the uBegOfRunCall flag set (see Analysis Hooks). Event 1 (Anal01) has been set up to read database values at the begin-of-run. It starts by reading data from the first database file defined by the DatabaseFile run control. Second and subsequent files are defined by either the DatabaseFile control settings or by the FILELIST database command. The FILELIST commands override the settings of the DatabaseFile control.

Data values can be read into either system blocks or other common blocks. System block variables are entered using the EGEOM keyword. An Excel file is used to define the structure of the system blocks. This variable management spreadsheet is used to define variable names, types, and locations. It also generates the FORTRAN code that defines the database internal structure. The database for the Proton Radiography experiment uses this management tool to provide arrays called IPRM, IIPRM, and RPRM for compatibility with legacy Q programs, though it is not necessary to use them to have database variables. These arrays have been equivalenced to the appropriate system blocks. The Excel file also generates LABEL commands for system block variables. Auto logging of database variables can be turned on at this point.

The structure a database file is simple. It consists of command lines and data lines. Command lines start in column 1. Data lines have a blank in column 1. Anything after an ! is a comment. Command lines have a keyword, sometimes a sub-keyword, then start and count numbers. Data lines can have one or more numbers on them separated by delimiters (usually blanks). Data lines immediately follow the appropriate command line. The default values depend on the variable. The only required keyword is RUN. This indicates which runs the following data applies too. Every file must have at least one RUN command line as the first command in the file. If two or more entries defined the same variable, the last definition is used. **THE ORDER OF APPEARANCE MATTERS.** Lines may only be 80 characters wide.

An ASCII log of the command and data lines used is written. When in DAQ mode, this file should be saved with the data to record the values actually read. It is called RUnnnn.log and is written to the PCDAQ directory.

If you are using the database to set values in the DAQ hardware (for example, pedestals), be sure to put the hardware setting code after the uBegOfRunCall to ANAL01 (Analysis Hooks). A good place for this is QRU2.

Database Example File

An example database file:

```
!
!  Example database file that shows how to control
!   the TDC gains over a range of runs.
!  This example shows two CAMAC TDCs in slots 3 and
!   4 of crate 2. It is assumed they are FERA modules
!   with assigned VSN (Virtual Station Numbers).
!   [Note: The interpretation of these three identification
!    numbers is actually up to the user.
!    This is simply one implementation.]
!
!    File: PCDAQHelp.doc
!    Author: Gary Hogan
!    Date: Aug. 22, 1996
!    Revision History:
!
```

```
!------------------------------
! Default definition.
!   This defines data from runs 0 to 0 which
!   the program interprets as meaning all runs.
!   Following data for specific runs will
!   overwrite these values.
!
RUN  0  0    ! Following data applies to all run numbers
GAINS   2   3   5   0   16     !  Crate 2, Slot 3, VSN 5,
                               ! Start sub-address 0, word count 16
  .1 .1 .1 .1   .1 .1 .1 .1
  .1 .1 .1 .1   .1 .1 .1 .1
GAINS   2   4   6   0   16     !  Crate 2, Slot 4, VSN 6,
                               ! Start sub-address 0, word count 16
  .1 .1 .1 .1   .1 .1 .1 .1
  .1 .1 .1 .1   .1 .1 .1 .1
!
!  New gains for runs 10-25. Only for sub addresses 0-3, Slot 3
!  Note, this will overwrite default gains.
!
RUN 10 25
GAINS  2 3 5 0 4
  .098 .097 .12 .105
!
!  Gains for Runs 26 to last run, slot 3
!
RUN 26 0
GAINS  2 3 5 0 4
  .095 .096 .111 .1076
!
!  Gains for Runs 10 to last run, slot 4
!   sub-addresses 0 -> 5
!
RUN 10 0
GAINS  2 4 6 0 6
  .085 .095 .131 .1276 .089 .075
```

Database Filenames

The program will search up to fifteen text files for database information. The first seven files can be defined using the DatabaseFile run control. If you need the remaining files, you need to use the FILELIST database command. The FILELIST command will override definitions from the DatabaseFile command. The FILELIST command allows you to define a branching tree structure for the database based on run numbers. Using this would allow you to read in only those files that actually pertain to the current run. This speeds up the read-in of data.

It is suggested that you use only the first DatabaseFile run control to define the top of the database files structure, then use the FILELIST command within the database to define subsequent file usage.

Database Filename Example

Define the top level database file using the DatabaseFile run control in your main script:

```
DatabaseFile=1,c:\MyDir\TopDataBaseFile.txt
```

Then in TopDataBaseFile.txt, begin the definition of tree structure:

```
Run  0 299   !  First year's data
FILELIST 2 1    ! File #2, one file
 c:\MyDir\Year1.txt

Run  300 600   !  Second year's data
FILELIST 2 1    ! File #2, one file
 c:\MyDir\Year2.txt
```

The Year1.txt and Year2.txt files can contain more FILELIST commands.

Database Keywords

10 keywords are presently recognized. Alphabetic case is not significant.

Table 4, Database Keywords

DIPS	Table to convert time differences into wire numbers.
DRIFT	Time average to distance tables.
EGEOM	Experimental geometry and other constants. Uses sub-keywords defined in the variable management spreadsheet. Data goes into the system blocks.
FILELIST	Define filename lists for input.
GAINS	Defines DAQ hardware to physics conversion.
LLIM	Lower limit for acceptable DAQ hardware data.
MAP	Defines DAQ hardware (ex., CAMAC) to detector mapping.
PEDS	Offsets to DAQ hardware data.
RUN	Defines run number range for following data.
ULIM	Upper limit for acceptable DAQ hardware data.

Keywords must be spelled out completely. No shortcuts are allowed.

The DIPS and DRIFT are specific keywords for Proton Radiography. Looking at the code for these (see CAL_RD.FOR and CAL_CONST.FI) will show you how to add new keywords to the database.

When you need to add new variables to the database, you would normally place them in the system blocks. This would mean updating the variable management spreadsheet with your new variable, then recompiling the DATABASE.DLL and USERROUTINES.DLL. If the new `data` will not easily fit into the system blocks, you will need to define new keywords to read in the data into new common blocks.

RUN

RUN is used to control which lines of the file are processed

Syntax:

```
RUN firstrun lastrun
```

The RUN keyword is followed on the same line by two numbers which are read as decimal (base 10). These are the minimum and maximum run numbers for which the following data applies (until the next RUN card). As a special case, 0 is satisfied by any run number. A RUN card should be first line in the file (except for comments).

RUN Examples

```
RUN 1567  1568      ! The following data applies to runs 1567 to 1568

RUN 1550  0         ! The following data applies to run 1550
                    ! and all runs thereafter.

RUN 0 0             ! The following data applies to all runs
```

MAP

MAP data is detector channel information. That is, it maps a given hardware channel (given by the information on the keyword line) into a physical detector channel.

Syntax:

```
MAP  crate  slot  VSN  first-channel   number-of-channels
       value  value … [Values in hexadecimal]
```

Crate, slot, and VSN are hardware identification numbers for a particular module. First-channel is the starting sub-address within the given module. Number-of-channels is a word count. Note, if Crate or Slot

numbers are proceeded by $, they are treated as hexadecimal numbers. Data values are assumed to be hexadecimal (no $ needed).

For Proton Radiography, the three hardware numbers are taken to refer to CAMAC crate number, slot number in the crate, and the FERA Virtual Station Number. For other experiments, the user may interpret them as they wish. What is important is that these three numbers are used to link the MAP information to information from the GAINS, PEDS, ULIM, and LLIM keywords. All three numbers must match for the proper linkage to be made.

The values following the MAP keyword defined how a specific DAQ hardware channel is to be mapped to a physical device. The interpretation of this value is left to the user. An example given below is from the early Proton Radiography experiments.

MAP Value

MAP values are 32-bit integer words that describe how to associate a given DAQ hardware channel to a physical device. In normal use, certain bits are designated as flags, other bits are detector numbers. Masks and shift counts used to decode the values are defined in ANA_CONST.FI. A value of -1 usually means a DAQ hardware channel is not connected to anything

An example of how to use the MAP values is given below from the Proton Radiography experiment.

```
bits 31-24: detector number. Defined values (hex):
       01 = Scintillator TDC
       02 = Scintillator ADC
       03 = Signal (latch) TDC
       04 = Signal (latch) ADC
       21 = Chamber TDC - U
       22 = Chamber TDC - D

bit 15: bad channel
       0 = normal value
       1 = data from this channel should be ignored

bits 14-00: element number -- wire or scintillator number. Possible values are 0 to
7FFF (0 to 32767 decimal)
```

MAP Examples

Example of single values:

Up-end of delay line chamber 4 = 22000004 (base 16)

Down-end of delay line chamber 5 = 21000005

Scintillator 2 TDC = 01000002

Mapping Crate 3, slot 5, VSN 8 into ADCs for scintillators 11 → 25. CAMAC sub-address 5 is undefined.

```
MAP 3 5 8 0 16
  0200000B  0200000C  0200000D  0200000E       ! scintillators 11 → 14
  0200000F                                     ! scintillator 15
  FFFFFFFF                                     ! sub-address 5 is not used
  02000010  02000011                           ! scintillators 16 → 17
  02000012  02000013  02000014  02000015       ! scintillators 18 → 21
  02000016  02000017  02000018  02000019       ! scintillators 22 → 25
```

MAP Programming Usage Notes

Programming notes from ANA_CONST.FI where the following arrays and parameters are defined.

```
Interpretation of the channel map:

There are four INTEGER*4 mapping arrays,
       MAP_10(-3:15,0:MAX_MAP_10) for 16-channel modules
       MAP_20(-3:31,0:MAX_MAP_20) for 32-channel modules
       MAP_60(-3:95,0:MAX_MAP_60) for 96-channel modules
```

```
       MAP_80(-3:127,0:MAX_MAP_80) for 128-channel modules
```

N_MAP_xx is the last row actually filled with mapping data in MAP_xx (xx = 10, 20, 60, or 80 [the number of words in hexadecimal]). Row 0 never contains valid mapping data and should be MAP_xx(*,0)=-1. The Nth row (where N <= N_MAP_xx <= MAX_MAP_xx)of MAP_xx contains the mapping for the hardware module given by:

```
       MAP_xx(-3,N) = FERA id (VSN)
       MAP_xx(-2,N) = Camac Slot
       MAP_xx(-1,N) = Camac Crate
```

The set of these three values is unique among all rows in MAP_xx. Then MAP_xx(0,N), MAP_xx(1,N), MAP_xx(2,N), ..., are the detector channels connected to module sub-address channels 0, 1, 2, ... for the given crate and slot. MAP_xx(y,N) = -1 means that no mapping information has been entered for channel y; otherwise, the value of MAP_xx(y,N) is interpreted as follows:

```
bits 31-24: detector number. Defined values (hex):
       01 = Scintillator TDC
       02 = Scintillator ADC
       03 = Signal (latch) TDC
       04 = Signal (latch) ADC
       21 = Chamber TDC - U
       22 = Chamber TDC - D

bit 15: bad channel
       0 = normal value
       1 = data from this channel should be ignored
```

bits 14-00: element number -- wire or scintillator number. Possible values are 0 to 7FFF (0 to 32767 decimal)

Example of single values:

```
       Up-end of delay line chamber 4 = 22000004  (base 16)

       Down-end of delay line chamber 5 = 21000005

       Scintillator 2 TDC = 01000002
```

There are three REAL*4 arrays of pedestal values (there are no 128-channel ADC's or TDC's):

```
       PED_10(0:15,0:MAX_MAP_10) for 16-channel modules
       PED_20(0:31,0:MAX_MAP_20) for 32-channel modules
       PED_60(0:95,0:MAX_MAP_60) for 96-channel modules
```

Three REAL*4 arrays of gain values,

```
       GAN_10(0:15,0:MAX_MAP_10) for 16-channel modules
       GAN_20(0:31,0:MAX_MAP_20) for 32-channel modules
       GAN_60(0:95,0:MAX_MAP_60) for 96-channel modules
```

Three INTEGER*2 arrays of lower window limits,

```
       LM1_10(0:15,0:MAX_MAP_10) for 16-channel modules
       LM1_20(0:31,0:MAX_MAP_20) for 32-channel modules
       LM1_60(0:95,0:MAX_MAP_60) for 96-channel modules
```

Three INTEGER*2 arrays of upper window limits,

```
       LM2_10(0:15,0:MAX_MAP_10) for 16-channel modules
       LM2_20(0:31,0:MAX_MAP_20) for 32-channel modules
       LM2_60(0:95,0:MAX_MAP_60) for 96-channel modules
```

These arrays are all indexed by Crate, Slot, and FERA VSN the same way as MAP_xx, but since Crate, Slot, and FERA VSN are actually stored only in the MAP_xx array, the placing of values into MAP_xx, PED_xx, GAN_xx, LM1_xx, and LM2_xx must be coordinated so that each Crate, Slot, and FERA VSN refers to the same row in all five arrays.

Examples

1. If Crate 5, Slot 4, VSN 8, sub-address 7 has its map value in MAP_10(7,19), then the corresponding pedestal is in PED_10(7,19) and gain is in GAN_10(7,19).
2. If you need to look up the lower limits for Crate 3, slot 5, VSN 9, you first loop from 1 to N_MAP_10 in MAP_10 looking for the these hardware identifier numbers. If you find a match at row 17, then the lower limits are in LM1_10(x,17).

Default values are:

Table 5, Default Database Values for MAP, etc.

item	default	variable
detector-channel map	-1	MAP_xx
pedestal	0.	PED_xx
gain	1.	GAN_xx
lower-limit	0	LM1_xx
upper-limit	32767	LM2_xx

PEDS

PEDS defines the pedestal zero offset for ADC's or the equal time offset for TDC's.

Syntax:

```
PEDS  crate  slot  VSN  first-channel  number-of-channels
        value  value …
```

See MAP for details on keyword values. If the number-of-channels is negative, only the first value is needed. It will be replicated for all the entries.

PEDS are stored as REAL*4 (F format--but you can omit the decimal point if there is no fraction). Default value is zero. A * for a value means do not change the value for that channel.

GAINS

GAINS define the multiplicitive conversion factors to convert ADC and TDC information into physics quantities.

Syntax:

```
GAINS  crate  slot  VSN  first-channel  number-of-channels
        value  value …
```

See MAP for details on keyword values. If the number-of-channels is negative, only the first value is needed. It will be replicated for all the entries.

GAINS are stored as REAL*4 (F format--but you can omit the decimal point if there is no fraction). Default value is one. A * for a value means do not change the value for that channel.

LLIM

These constants are currently used for sparcification such as for LeCroy ADCs. LLIM is the lower bound, ULIM is the upper bound.

Syntax:

```
LLIM  crate  slot  VSN  first-channel  number-of-channels
        value  value …
```

See MAP for details on keyword values. If the number-of-channels is negative, only the first value is needed. It will be replicated for all the entries.

LLIM values are stored as 16-bit integers. Default value is zero. A * for a value means do not change the value for that channel.

ULIM

These constants are currently used for sparcification such as for LeCroy ADCs. LLIM is the lower bound, ULIM is the upper bound.

Syntax:

```
ULIM  crate  slot  VSN  first-channel  number-of-channels
```

```
            value  value …
```

See MAP for details on keyword values. If the number-of-channels is negative, only the first value is needed. It will be replicated for all the entries.

ULIM values are stored as 16-bit integers. Default value is zero. A * for a value means do not change the value for that channel.

EGEOM

EGEOM defines **E**xperimental **GEOM**etry variables.

Syntax:

```
EGEOM  sub-keyword  n1  n2
        value  value …
```

The default values for n1 and n2 is 1. For scalar quantities the numbers have no meaning. Otherwise, n1 is the array starting index, and n2 is the number of values given. 2-D and 3D arrays should be treated as 1-D arrays in the geometry file. Values may be integer or real, depending on the sub-keyword. The * place holder has no meaning. Assuming you have used the EXCEL variable management tool, the sub-keyword is the same as the variable name. The data entered here will reside in the appropriate system_block. This is the only keyword that interacts with the system blocks. All other database keywords put their data in other common blocks. Sub-keywords must be spelled out completely. For the Proton Radiography experiments, most of the time this keyword is used to fill variables that have been assigned to the IPRM, IIPRM, or RPRM arrays. Negative counts for n2 do not work with this keyword.

The structure of the EGEOM sub-keywords are defined in the block data file EXPER_GEOM.FOR which is generated by the variable management spreadsheet.

EGEOM Examples

Example of how to fill database values. Variable names are made up just for this example.

Filling scalar variables Zposition and MaxEnergy

```
EGEOM Zposition  1  1
  199
EGEOM MaxEnergy   ! n1 and n2 are optional for scalars
  52.8
```

Filling an array WirePos(10).

```
EGEOM WirePos  1  10
  1 2 3 4 5 6 7 8 9 10
```

Change WirePos(7)

```
EGEOM WirePos  7  1
      7.1
```

Change WirePos(4) and (5)

```
EGEOM WirePos  4  2
   4.1  5.1
```

Filling array Gtherm(2,3)

```
EGEOM Gtherm 1 6
    5  7     8  9    3.4  9.8
```

Changing Gtherm(2,1)

```
EGEOM Gthrem 3 1
   8.1
```

Changing Gtherm (1,3) and (2,3)

```
EGEOM Gtherm  5  2
     3.5  9.9
```

FILELIST

FILELIST is used to redefine the list of input files using the run number as the key

Syntax:

```
FILELIST  First-element  Number-of-elements
   filename
   filename
   …
```

Where:

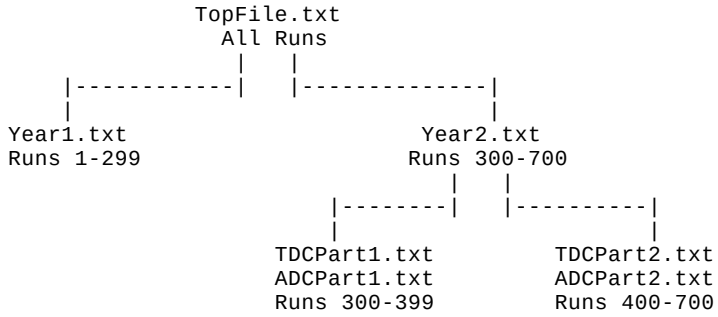
First-element : Location in file list to substitute first entry

Number-of-elements : Number of entries that follow

Each filename is on a separate line. Remember to indent one space at the beginning of the line. Filename lengths are limited to 79 characters (the length of the input line). File names are substituted into the database file list. If that entry in the file list has already been read or is the currently opened file, the substitution has no effect.

FILELIST example

Assume you want to have the following branch structure to the database:



For various run numbers, you would read in

Run Number	**50**	**310**	**450**	**900**
File #1	`TopFile.txt`	`TopFile.txt`	`TopFile.txt`	`TopFile.txt`
File #2	`Year1.txt`	`Year2.txt`	`Year2.txt`	
File #3		`TDCPart1.txt`	`TDCPart2.txt`	
File #4		`ADCPart1.txt`	`ADCPart2.txt`	

Table 6, FileList Example

Then you should have the following FILELIST commands in various files:

METHOD 1. Structure defined throughout files

Define the top level database file using the DatabaseFile run control in your main script:

```
DatabaseFile=1,c:\MyDir\TopFile.txt
```

Somewhere in TopFile.txt you would have:

```
RUN 1  299
FILELIST 2 1   ! Define the second database file
   c:\MyDir\YEAR1.TXT

RUN 300 700
FILELIST 2 1   ! Define the second database file
   c:\MyDir\YEAR2.TXT
```

and in YEAR2.TXT, you would have:

```
RUN 300 399
FILELIST 3 2    ! Define the third and fourth database files
   c:\MyDir\TDCPart1.txt
   c:\MyDir\ADCPart1.txt

RUN 400 700
FILELIST 3 2
   c:\MyDir\TDCPart2.txt
   c:\MyDir\ADCPart2.txt
```

METHOD 2. Structure defined in one place

Define the top level database file using the DatabaseFile run control in your main script:

```
DatabaseFile=1,c:\MyDir\TopFile.txt
```

Somewhere in TopFile.txt you would have:

```
RUN 1  299
FILELIST 2 1   ! Define the second database file
   c:\MyDir\YEAR1.TXT

! Define the second through fourth database files
RUN 300 399
FILELIST 2 3
   c:\MyDir\YEAR2.TXT
   c:\MyDir\TDCPart1.txt
   c:\MyDir\ADCPart1.txt

RUN 400 700
FILELIST 2 3
   c:\MyDir\YEAR2.TXT
   c:\MyDir\TDCPart2.txt
   c:\MyDir\ADCPart2.txt
```

DIPS

This defines lookup tables for converting delay line differences into wire numbers. It is specific to the early Proton Radiography experiments.

Syntax:

```
DIPS  chamber  first-channel  number-of-channels
        value  value …
```

DIPS values are stored as 16-bit integers. Default value is zero (meaning undefined wire). Variables are defined as a separate common block in CAL_CONST.FI

DRIFT

This defines lookup tables for converting delay line averages into drift distances. It is specific to the early Proton Radiography experiments.

Syntax:

```
Drift  chamber  wire-number  first-channel  number-of-channels
        value  value …
```

DRIFT values are stored as 16-bit integers. Default value is zero. If wire-number is zero, the data is assumed to be for all wires and stored in DriftConstants instead of DriftWireConstants. Variables are defined as a separate common block in CAL_CONST.FI

Database Syntax

Items on a line are separated by one or more spaces, tabs, or commas, or some combination thereof. Note this means you CANNOT use double commas to denote omission of an item. Omitted data values can be

indicated by entering an asterisk (*) as the item in the corresponding position. The * placeholder has no meaning for the geometry values (EGEOM).

Keywords MUST start in the first column; data must NOT start in the first column. Anything after an exclamation point is ignored. This is for inserting comments. Lines may only be 80 columns wide.

Variable ManagementThere is an EXCEL spreadsheet available to help with managing variables in the system blocks. There are currently three system blocks defined for PCDAQ. They are a 16-bit integer array, a 32-bit integer array, and a 32-bit floating point array. Different parts of these arrays are used for a variety of purposes: I/O buffers, database values, and analysis results. Not only are there a large number of type and equivalence statements that must be written up, but also LABEL and database structures need to be defined in order to get the most out system block usage. The variable management spreadsheet simplifies this by providing a single spreadsheet in which the user defines how the array usage in the system blocks is to be parsed out. Once the definition is complete, a macro in the spreadsheet generates the appropriate FORTRAN code and LABEL commands.

The spreadsheet currently used for variable management is called PARM2.XLS and it sits in the Database directory. [Tip: create an icon or shortcut to this file.] It produces three files: DATABASELABEL.SCR, EGEOM.FI, and EXPER_GEOM.FOR. DATABASELABEL.SCR is script file defining the LABEL commands describing the system blocks. EGEOM.FI is a FORTRAN include file that has the type and equivalence statements for the system blocks. EXPER_GEOM.FOR is a FORTRAN block data program that defines the structure of the EGEOM keyword portion of the database. These files are generated when the user clicks on GENERATE under the Tools menu. Once the files are generated, the user needs to recompile the Database and UserRoutines DLLs and then copy the DLLs to the system directory.

To define a new variable in the system block, you first insert a new row in the spreadsheet where you want the variable definition to be placed. Then you enter the following information about the variable:

Name: This is the variable's FORTRAN name. It should correspond to the FORTRAN naming conventions. The type of the variable is derived from the system block in which it resides. This name is the sub-keyword that should be use on the EGEOM line for setting the value in the database. The name may be up to 20 characters long. Avoid using the characters "_^@" as they are special formatting controls in the plotting package

Dim1, Dim2, Dim3: The dimension(s) of the variable. If the dimension is 1, that dimension is ignored. Thus a simple scalar variable would be DIM1 = 1, DIM2 = 1, DIM3 = 1. A 1-D array of size n would be DIM1 = n, DIM2 = 1, DIM3 = 1. A 2-D array of n x m would be DIM1 = n, DIM2 = m, DIM3 = 1. Arrays subscripts start with 1.

Array: A 1-D array to equivalence this variable too. Variable equivalencies can be nested as deep as FORTRAN allows, that is, you can equivalence a variable to an array that is equivalenced to an array that is equivalenced to an array, etc. The only condition is that you must ultimately reach one of the system block arrays (IntegerData, SingleData, or LongData).

Index: Where in the array the variable is to be equivalenced.

Event: The event number the variable should be associated with on the LABEL command.

Log: Flag that the auto-logging flag should be set on the Label command. Logging is set if this is 1, not set if it is 0.

Form: The format command to use on the label command.

Comments: Comment describing the variable. This will also be used as the name on the Label command and will appear on the Data Screen window. So try to keep this short. Only the first 50 characters are saved in the EXPER_GEOM.FOR data structure statements.

Once you have added or changed the variables you want, generate new versions of the system block files as described above.

The LABEL command generated has as it's index the location of the variable in the system block. This makes this line a good starting point for defining a histogram for a variable.

Database Header Files

There are two header files, EGEOM.HEAD and EXPER_GEOM.HEAD that the management program uses when generating EGEOM.FI and EXPER-GEOM.FOR. These files contain the code that goes before the lines generated by the spreadsheet itself. The line in EGEOM.FI that defines the maximum number of variables will be changed by the spreadsheet as needed.

Using the Database in Other Programs.

The database programs are in a separate DLL. With only minor modifications, the code can be used in other programs. Please see the author for details or look in ANAL01. Important routines are:

CALL SET_CONST_UNITS(data-in, log-out)
where log-out is the unit number for writing the log file (defaults to 6) and data-in is the unit number for reading the constants file (default 33). Open the log file explicitly if you want it to have a nice name.

status = SET_CONST_FILES(file-list, number-of-files)
file-list is a character ARRAY containing the file names. The files are read in the order of the array elements. Function is I*4. Value = 0 if list is okay.

ISTAT=MEGA_CONST(run-number)
Reads the data files and fills the constants arrays appropriately for the run-number. It reads the files only if the run-number has changed since it was last called (exceptions: (1) using a negative run-number will always cause the file to be read; (2) calling SET_CONST_FILES will force reloading of the constants on the next call MEGA_CONST, regardless of the run number). ISTAT=1,2,3 for warnings, errors, or both. Search the log file for *W* and *E* to find them. Otherwise, ISTAT=0

EGEOM.FI and EXPER_GEOM.FOR are created by the variable management spreadsheet. If you are going to use this DLL in another program, you may need to change the header files EGEOM.HEAD and EXPER_GEOM.HEAD to reflect you own common block usage. PCDAQBuffersImport.inc and PCDAQBuffers.inc would also need to be replaced. (These define the system block commons).

DATABASE.FI. An include file that brings together the various include files in the database except for the system blocks. It also contains the compiler attribute statements needed to properly link to the DLL common blocks.

DATABASEIMPORT.FI Defines the interface to the database routines.

PCDAQBuffers.inc and PCDAQBuffersimport.inc are needed because that is where the system block common blocks are defined. If you were writing a new program, you would presumably move these common blocks elsewhere.

Database File Updates

Database files can be changed with a text editor (ex., Notepad) or by appending files to existing files. One convenient method is to define an icon or shortcut symbol to be the Notepad editor with the database file as the input file.

Writing Database Files

To make writing flat files a relatively painless process, subroutines with names of the form CAPND_* are available to create and format data files. At this time, however, there seems to be a problem with using them within other DLLs.

Run Numbers for Output

A RUN card should be the first line in the file (except for comments). In setting "firstrun" and "lastrun", consider that this file may be designed to be appended to the END of another flat file. Normally, "firstrun"

should be the run number of the data from which the constants were determined. If it is set smaller than this, you must be confident that the results apply to prior runs as well, and be certain that it will not overwrite values accurately known for those runs. If the constants are current, "lastrun" must be set to 0, since you cannot anticipate for how many runs they will remain valid. If, however, they have been superseded by current conditions (e.g., they are tardy in production, or are corrections to old values), then an accurate "lastrun" must be supplied to avoid their overwriting the current values when the flat file is read.

Database Output Subroutines

1. CAPND_CREATE(lun, filename) -- creates the file
   where lun = I*4 FORTRAN unit number (you choose to avoid conflicts)
   filename = CHARACTER name for the file
   (by default, this file is created in your default directory;
   default name is MEGA_CAPND, default extension is .TXT)

2. CAPND_ALL (firstrun, lastrun, lun, keyword-list, list-length)
   Master dump routine. Can be use to dump the entire contents of the database or all the data under selected keywords.
   Firstrun = I*4, starting run number to use in RUN keyword
   Lastrun = I*4, last run number to use in RUN keyword
   lun = I*4, logical unit number of output file. File must already be open.
   Keyword-list = Character array giving list of keywords to dump.
   Case not important. A value of 'ALL' will result in
   all keywords being dumped. For this option,
   list-length must be 1.
   List-length = I*4, Number of items in keyword list.

3. CAPND_RUN(firstrun,lastrun)
   writes RUN keyword line
   firstrun = I*4 run number, first run for which following is valid
   lastrun = last run for which following is valid (normally, this will not be known, so 0 is used -- meaning that the data applies to ALL runs following firstrun until it is overwritten by a later entry)

4. CAPND_MAP(crate, slot, VSN, firstchan, numberchan) -- writes MAP keyword line
   CAPND_PEDS(crate, slot, VSN, firstchan, numberchan) -- writes PEDS keyword line
   CAPND_GAINS(crate, slot, VSN, firstchan, numberchan)-- writes GAINS keyword line
   CAPND_LLIM(crate, slot, VSN, firstchan, numberchan) -- writes LLIM keyword line
   CAPND_ULIM(crate, slot, VSN, firstchan, numberchan) -- writes ULIM keyword line
   crate: CAMAC crate
   slot = slot in hardware crate
   VSN = virtual station number
   firstchan = module channel number corresponding to first of the following data values (first channel in the MODULE is always numbered 0)
   numberchan = number of data values following.

   Note, these routines only write the keyword. To write the data, use the write routines below.

5. CAPND_DIPS (chamber, first-wire, number-of-wires)
   Writes keyword line only. To write the data, use the write routines below.

6. CAPND_DRIFT (chamber, wire-number, first-value, number-of-values)
   Writes keyword line only. To write the data, use the write routines below.

7. CAPND_EGEOM(variable-list, number-of-variables)
   Writes EGEOM keyword and data.
   variable-list is a CHARACTER*15 list of sub-keywords giving which variables to write out.
   Number-of-variables is an I*4 word giving the length of the variable list.
   If number-of-variables = 1 and variable-list(1) = "ALL", then all the variables are written out.

Variable names are case independent. Values of variables come from common block. If the variable is an array, the entire array is written out.

8. CAPND_FILELIST - Write out current input file list. No parameters.

9. CAPND_R4DATA(array, number) -- writes formatted lines of REAL*4 data
CAPND_I4DATA(array, number) -- writes formatted lines of INTEGER*4 data
CAPND_Z4DATA(array, number) -- writes formatted lines of INTEGER*4 data in hex format
CAPND_I2DATA(array, number) -- writes formatted lines of INTEGER*2 data
array = (R*4, I*4, or I*2, as appropriate) array containing the data to be written
number = number of values to be written out
SPECIAL CASE: to get the asterisk placeholder, put the following values in the corresponding array element:
-1.E30 for CAPND_R4DATA
-32767 for CAPND_I2DATA (='8001'X)
-2147483647 for CAPND_I4DATA (='8000001'X)

Database Output Example

As a simple example, suppose you have determined TDC offsets and slopes for two 16-channel modules with crates ID(1) and ID(2), slots GA(1) and GA(2), and placed them in the first two rows of REAL arrays OFF(16,*) and SLO(16,*), and that these values were determined with data from run number NRUN. Then the following code will produce a file DatabaseOutput.TXT suitable for appending to the database:

```
INCLUDE 'PCDAQBUFFERS.INC'
INCLUDE 'PCDAQBUFFERSIMPORT.INC'
INCLUDE 'DATABASE.FI'
INCLUDE 'DATABASEIMPORT.FI'
…

CALL CAPND_CREATE(35,'DatabaseOutput.TXT')
CALL CAPND_RUN(NRUN,0)
DO I=1,2
   CALL CAPND_PEDS(ID(I),GA(I),0,0,16)
   CALL CAPND_R4DATA(OFF(1,I),16)
   CALL CAPND_GAINS(ID(I),GA(I),0,0,16)
   CALL CAPND_R4DATA(SLO(1,I),16)
END DO
```

Analysis Hooks

The control program calls a number of user definable programs. They are defined in the FORTRAN DLL library "UserRoutines" and in the vbAnalxx modules in the DAQControl project. The user can place code in either or both routines. The Visual Basic version is called before the Fortran version. Empty versions of the programs are provided. The central program(s) that the user needs to provide is the event analysis routine ANALnn, where nn is the event number. For example, ANAL06 is the analysis program for event 6. The current range of allowed events is 1 to 24. The calling procedure for the analysis program is:

ReturnCode = ANALnn (CallType)

The analysis routine is called at a number of points in the control program. The value of CallType tells the user what is going on. Parameters defining the call types are provided in the include file PCDAQBuffers.inc.

The call type parameters are:

uNewCall :: Data in input buffer has changed (new CAMAC read done). Buffer pointers should be reset, then event analysis done. Both UserEnable and EventEnable must be set for the call to occur. Called if hardware or software trigger occurs.

uRecursiveCall :: Program is being called for second (or more) times for a given hardware input event. Do appropriate event analysis. Both UserEnable and EventEnable must be set for the call to occur. Called if recursive request or software trigger occurs.

uBegOfRunCall :: Analysis is being called at start of run. Use to initialize program or define data to dump to tape at start of run. No data has been read at this point. Only EventEenable must be set for call to occur. Taping of labeled variables for event will occur after all events are called if uAbortOuput is not set.

uEndOfRunCall :: Analysis is being called at end of run. Use to define data to be dumped to tape at end of run. Only EventEnable must be set for call to occur. Taping of labeled variables for event will occur after all events are called if uAbortOuput is not set.

uClearCall :: Garbage clean up at end of each analysis loop. Called after automatic histogramming and taping are done. Note, variables nulled out here will not show up on the data screen. Both UserEnable and EventEnable must be set for call to occur. Called after every uNewCall or uRecursiveCall is made or if analysis is skipped due to May Process test.

uGetTrigger :: Do CAMAC input code. This will usually be a set of FCNA or GPIB calls. Data should go into appropriate system blocks as defined by CAMACnn label commands. Pointers to the system blocks are in the common block /EventDefinitionCommon/ located near the end of PCDAQBuffers.inc. The user is responsible for setting the event size array(s) so the control program can properly output the raw data. This is the equivalent of the "QAL" code for an event in the Q system.

In the VB analysis code, used to check for an Active X Sever Trigger or check for a completed asynchronous read. Change return result to 0 if trigger found. Specifically, setting uAbortOutput means that a trigger was not found. If you want to kill output when a trigger is found, use the Fortran Analnn function to set uAbortOutput.

uGetMonteCarlo :: Generate a Monte Carlo event. Only the Fortran version is called.

uBegOfSessionCall :: Called just before hardware connections are made. For instance, connections to other ActiveX servers by VB programs can go here (though uBegofBatchCall may be a better place).

uEndOfSessionCall :: Called just after hardware connections are broken. For instance, disconnections to other ActiveX servers by VB programs must go here if not done in uEndofBatchCall.

uBegOfBatchCall :: Called at the beginning of a group of runs. For live data, the start of data taking when rollover is enabled. A good place to verify connection to Active X servers, and find out

what is actually connected. Also possibly, to freeze controls on servers (for example, if they are using GPIB). Clear out variables that accumulate over multiple runs. Called only once when in DAQ and rollover mode.

uEndOfBatchCall :: The end of a batch of runs. Put analysis of multi-run data here.

uBegOfSessionWrite :: User write is being called at start of session. Not implemented.

uEndOfSessionWrite :: User write is being called at end of session. Not implemented.

uBegOfBatchWrite :: User write is being called at start of batch. Not implemented.

uEndOfBatchWrite ::User write is being called at end of batch. Not implemented.

uBegOfRunWrite :: User write is being called at start of run. Not implemented.

uEndOfRunWrite :: User write is being called at end of run. Not implemented.

uDataWrite :: Fortran: User write is being called at end of each event analysis. Not implemented. VB: Write data to Active X Servers in replay mode at the end each event analysis

uBegOfSessionRead :: User Read is being called at start of session. Not implemented.

uEndOfSessionRead :: User Read is being called at end of session. Not implemented.

uBegOfBatchRead :: User Read is being called at start of batch. Not implemented.

uEndOfBatchRead :: User Read is being called at end of batch. Not implemented.

uBegOfRunRead :: User Read is being called at start of run. Not implemented.

uEndOfRunRead :: User Read is being called at end of run. Not implemented.

uDataRead :: Fortran: User Read is being called at start of analysis. Not implemented. VB: Get data from Active X Servers. DAQ or Monte Carlo modes only?

The value of ReturnCode tells the control program what to do next. This return code is a set of control bits and can be combined. Bit definitions are:

uRecursiveRequest :: Requests that this program be called again.

uNoSubTriggers :: Do not issue software triggers that depend on this event as the trigger source.

uAbortOutput :: Do not tape output from this event. Histogram fill status is not affected.

uAbortHistograms :: Do not fill histograms from this event. “Taping” status is not affected.

uAbortAll :: If doing analysis (uNewCall, uRecursiveCall, or uEndOfRunCall), all further work on this trigger is aborted, including histogram fills, taping, software triggers, and outstanding recursive requests. The program then waits for the next hardware trigger. If it happens on the CAMAC read call (uGetTrigger) or the begin-of-run call (uBegOfRunCall), something has gone badly wrong and the run is stopped immediately.

uSuspendRun :: Suspend run request.

uAbortHistograms :: Do not fill histograms from event

uAbortRun :: Abort the run. This brings the run to an immediate halt.

uErrorMessage :: Raise an error message. In Visual Basic code, use the variables AnalErrorNumber and AnalErrorString to store your error message. In Fortran, use ErrNumber and ErrString.

uEndRun :: Cause the run to end normally, i.e., with calls to end-of-run routines.

uRolloverRun :: Cause the run to end normally, i.e., with calls to end-of-run routines. Then immediately start a new run without user intervention.

Other routines are provided. They have the calling procedure:

ReturnCode = UserRoutine(EventEnableFlag)

EventEnableFlag is an array containing the state of the Event Enable Flags. If EventEnableFlag(n) = vbChecked, then event n is enabled, otherwise not. Users are on their honor not to change the state of this variable. If the return code is zero, an error is assumed and the run terminates. The user defined routines are called as follows. User routines are in capital letters:

Initialization ("qstart"):

ReturnCode = **ANALnn**(uBegOfSessionCall).
Setup CAMAC.
Test event module location.
Check thatdatabase files listed on control form exist.
ReturnCode = **USERSTART**( ).

Run Start:

Null out system blocks.
Clear Histograms.
Check that database files listed on control form exist.
ReturnCode = **ANALnn**(uBegOfBatchCall). ANAL01 is the read.
Reset CAMAC (ensure event module not enabled).
Open output file and dump begin of run information.
ReturnCode = **XSRUN**(EventEnable) or **QSRUN**(EventEnable).
ReturnCode = **QRU1**(EventEnable).
Enable event module.
Automatic histogram clear if checked.
ReturnCode = **ANALnn**(uBegOfRunCall). ANAL01 is the read.
Dump by-event begin-of-run information.
ReturnCode = **QRU2**(EventEnable).

Data Loop

Loop until run ends or suspends.
ReturnCode = **ANALnn**(uGetTrigger) or **ANALnn**(uGetMonteCarlo).
Loop until recursion done or new data.
ReturnCode = **ANALnn**(uDataRead)
Loop until all software triggers are done.
ReturnCode = **ANALnn**(uNewCall) or **ANALnn**(uRecursiveCall).
Simple linear transforms.
SIMPLEUSER1(input value, slope, offset)
SIMPLEUSER2(input value, slope, offset)
End software trigger loop.
ReturnCode = **ANALnn**(uDataWrite)
Automatic fill histograms.
Automatic output to tape.
ReturnCode = **ANALnn**(uClearCall) if **ANALnn** called in above loop.
End loop.
End loop.

Suspend Run

ReturnCode = **QSPND**(EventEnable).
ReturnCode = **QSPND1**(EventEnable).
Disable event module.
ReturnCode = **QSPND2**(EventEnable).
Write suspend marker to tape.

Resume Run

ReturnCode = **QRESUME**(EventEnable).
ReturnCode = **QRESUME1**(EventEnable).
Enable event module.
ReturnCode = **QRESUME2**(EventEnable).
Write resume marker to tape.

End Run

ReturnCode = **XERUN**(EventEnable) or **QERUN**(EventEnable).
ReturnCode = **QER1**(EventEnable).
Disable event module.
ReturnCode = **QER2**(EventEnable).
ReturnCode = **ANALnn**(uEndOfRunCall). Put histogram analysis code here.
ReturnCode = **ANALnn**(uEndOfBatchCall). Put histogram analysis code here.
Dump by-event end-of-run information.
Automatic histogram dump if checked.
Close output file.

Disconnect (“Qstop”)

Call **USERSTOP** ( ).
Disconnect from CAMAC and GPIB.
ReturnCode = **ANALnn**(uEndOfSessionCall).

Output Format

Output files are binary files. Records come in two general flavors. The first is a header/descriptor record. All headers have the same structure and size. What the entries in a given header mean varies by header type. The second is a variable length data buffer that is described by the proceeding header record. In words, an file might look like the following. Headers are surrounded by periods, data buffers are in italic. Headers and buffers that are not needed are not taped. A detailed description of output header formats can be found in OutputFormat.xls.

Output Example

.Begin of Run Header.

.Begin Definition of Labels.
.Label description.
Buffer containing label information.
.Label description.
Buffer containing label information.
…
.End Label Description.

.Begin Definition of output buffers.
.Description of Fixed output buffers starting index, type Integer.
*List of pointers to integer (I*2) data, all events.*
.Description of Fixed output buffers ending index, type Integer.
*List of pointers to integer (I*2) data, all events.*
.Description of Fixed output buffers starting index, type Single.
*List of pointers to single (R*4) data, all events.*
.Description of Fixed output buffers ending index, type Single.
*List of pointers to single (R*4) data, all events.*
.Description of Fixed output buffers starting index, type Long.
*List of pointers to long (I*4) data, all events.*
.Description of Fixed output buffers ending index, type Long.
*List of pointers to long (I*4) data, all events.*
.Description of CAMACnn output buffer, type Integer, event nn.
.Description of CAMACnn output buffer, type Single, event nn.
.Description of CAMACnn output buffer, type Long, event nn.
…
.End Definition of output buffers.

.Begin Control Flag Dump.
.Event Enable Flags.
Enable flag buffer.
.Event Source Flags.
Enable source buffer.
…
.End Control Flag Dump

.Begin Begin-Of-Run dump.
.Begin Event 1 output.
.ANAL01 results, begin of run, integer.
Integer data buffer
.ANAL01 results, begin of run, single.
Single data buffer
.ANAL01 results, begin of run, long.
Long data buffer

.End Event 1 output
….
.End Begin-Of-Run dump

.Begin Data Taking.
.Begin hardware event nn.
.Begin recursion loop 1.
.Begin Event 1 output.
.CAMAC01 data, integer
Integer data buffer
.CAMAC01 data, single
Single data buffer
.CAMAC01 data, long
Long data buffer
.ANAL01 results, integer.
Integer data buffer
.ANAL01 results, single.
Single data buffer
.ANAL01 results, long.
Long data buffer
.End Event 1 output
.Begin Event 2 output.
….
.End Event 2 output
.End recursion loop 1.
.Begin recursion loop 2.
….
.End recursion loop 2.
.End hardware event nn.
….
.Suspend run marker.
.Resume run marker.
.End data taking.

.Begin End-Of-Run dump.
.Begin Event 1 output.
.ANAL01 results, end of run, integer.
Integer data buffer
.ANAL01 results, end of run, single.
Single data buffer
.ANAL01 results, end of run, long.
Long data buffer
.End Event 1 output
….
.Event nn Final Counters.
…
.End End-Of-Run dump

.Begin Dump of Histograms.
.Begin histogram nn.
. Histogram description.
Buffer containing histogram description.
.Histogram Data.
Buffer containing histogram data (1-D or 2-D).
.Histogram Time Dates.
Buffer containing histogram dates (Time Hist, once per dump).
.Histogram Time Index.
Buffer containing index to last entry (Time Hist, once per dump).

.Histogram Time Data.
Buffer containing histogram data (Time Hist).
.End histogram nn
….
.End Histogram Dump.

.ASCII Comment.
Buffer containing ASCII comment.

.End of run header.

Header Format

The header record has the following format:

Time Stamp (R*8). Microsoft Date, which I think is the Julian time to the nearest hundredth second in units of days.

Record Type (I*4).

Record Sub Type (I*4).

Length of Following Buffer [bytes] (I*4).

Other information 1 (I*4).

Other information 2 (I*4).

See the spreadsheet “OutputFormat.xls” for detail definitions.